Fuzzing on the Openflow Protocol

Sébastien Coorevits

Internship report

Openflow Protocol

Sébastien Coorevits

Year 2015–2016

Internship conducted for the team MADYNES of the INRIA

Supervisor : Jérome François

Report Academic Supervisor : Thibault Cholez

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Coorevits, Sébastien

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31417010

Année universitaire : 2015–2016

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Fuzzing on the Openflow Protocol

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 18 août 2016

Signature : Sébastien COOREVITS

Contents

Contents iii

1 Introduction 1

2 Presentation of the research laboratory 3

2.1 Presentation of the LORIA . . . 3

2.1.1 General organisation . . . 3

2.2 Presentation of the inria . . . 4

2.3 Presentation of the inria Nancy - Grand EST . . . 5

2.4 Presentation of the MADYNES team . . . 5

3 State of the art 7

3.1 Traditional routing . . . 7

3.2 Software Defined Network . . . 7

3.3 OpenFlow Protocol . . . 8

3.3.1 Flows . . . 8

3.3.2 Matching . . . 9

3.3.3 Working of Openflow . . . 9

3.4 Device . . . 10

3.5 OFtest . . . 11

3.5.1 Structure of OFtest . . . 11

3.6 Detailled problem statement . . . 12

4 Documentation and corrections of OFtest 13

4.1 Documentation . . . 13

4.1.1 Structure of the documentation . . . 13

4.1.2 basic . . . 14

4.2 Corrections of OFtests . . . 15

4.2.1 Approach . . . 15

4.2.2 Type of errors and corrections . . . 15

4.2.3 Ethernet source matching 16

5 Fuzzing **17**

5.1 Introduction to Fuzzing 17

5.2 Implementation 17

5.3 Testing 18

6 Conclusion **19**

bibliography / webography **20**

List of Figures **21**

Résumé **23**

Abstract **23**

A APPENDIX **24**

1 Introduction

I have conducted my second year internship from June the 6th of 2016 to July the 29th of 2016 in the research team MADYNES at Inria. The internship was supervised by Mr Jérôme FRANCOIS member of the MADYNES team.

The goal for this internship was to develop a documentation for the test library OFTEST, this library allows to test an OPENFLOW switch within a controlled environment and then to developed a fuzzer. The first part includes making the switch take all the tests of the library and then documenting their effects and results, when it was possible, I had to correct bugged tests. I also had to document any bugs of the switch or openflow that could be found during those tests. The second part of the internship was to create a fuzzer aiming at using the protocol in a non standard manner, with or without corrupted request and to monitor the behaviour of the switch using the OFTEST library.

I would like to thanks Mr Jérôme FRANCOIS for the supervision and his answer to my questions, and Mr Thibault CHOLEZ for his help and conviviality during the internship.

2 Presentation of the research laboratory

2.1 Presentation of the LORIA

The LORIA, standing for Laboratoire Lorrain de Recherche en Informatique et ses Applications, is a joint research unit between three groups. Those groups are the CNRS, the University of Lorraine and the inria. The laboratory was created in 1997 and is tasked with fundamental and applied research in computer science. The LORIA is structured in 30 teams, 15 of them are joint with the inria, divided in five departments. This is one of the largest research laboratory of Lorraine with more than 450 people.

2.1.1 General organisation

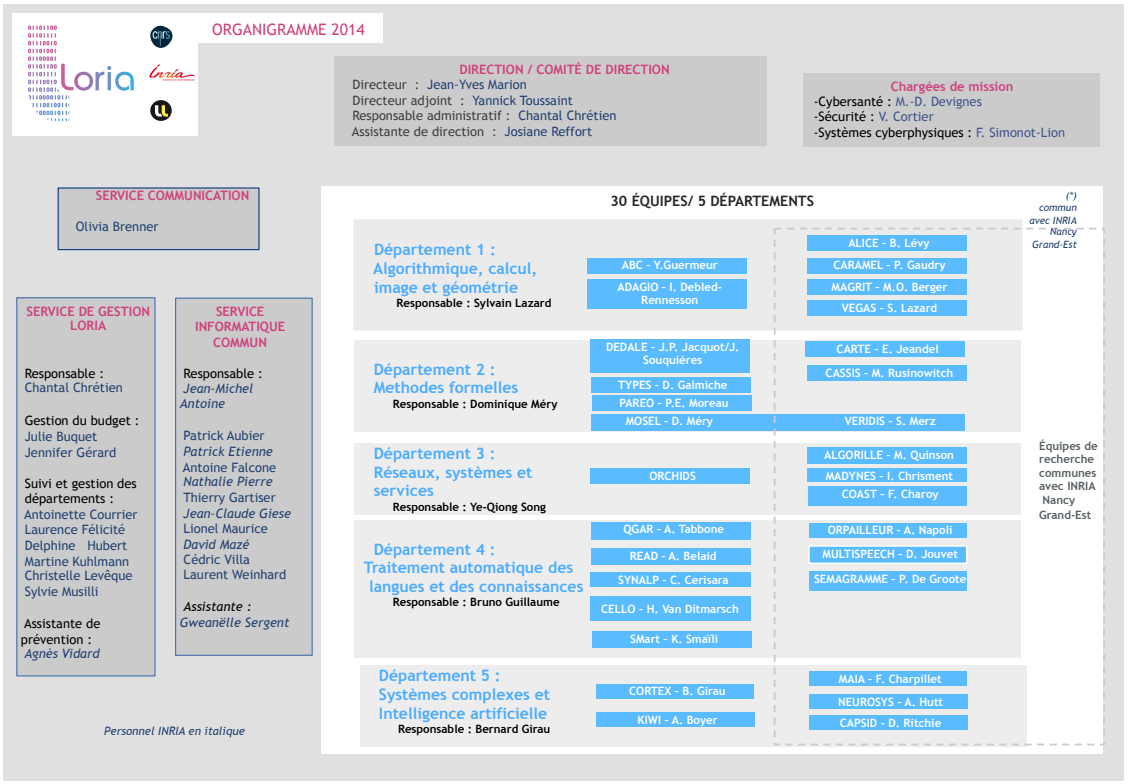

Figure 2.1 – general organisation of the LORIA (www.loria.fr/en)

2.2 Presentation of the inria

Inria is the French Institute for Research in Computer Science and Automation. The institute was created in 1967 at Rocquencourt near Paris, it is tasked with research in computer science and mathematics. It is divided into 8 centers (Bordeaux, Grenoble, Lille, Nancy, Paris-Rocquencourt, Rennes, Saclay, and Sophia Antipolis).

Some key facts :

- Over 2700 researchers
- €235 million (2013)
- run by Antoine PETIT

Figure 2.2 – general organisation of the inria (www.inria.fr/)

2.3 Presentation of the inria Nancy - Grand EST

This center was created in 1986, it has developped strong ties with the University of Lorraine and the CNRS. There are 24 teams with a total of more than 170 researchers.

Figure 2.3 – inria (http://www.pss-archi.eu/)

2.4 Presentation of the MADYNES team

This resarch TEAM, currently led by Isabelle CHRISMENT, is located in the premise of the LORIA and inria, it is focused on networks, system and service, and Distributed computing. It adresses two areas in particular which are Autonomus management and Functionnal Areas.

3 State of the art

3.1 Traditional routing

Traditional routers or switches have both the data plane and the control plane on the same device. The data plane is responsible for forwarding the packets whereas the control path is the routing algorithms in place to chose where to send the packet.

Traditionally each vendor has his own interface, the internal working of the switch is hidden and differs from one constructor to an other. This makes it quite difficult for researchers to test new ideas, new network protocols and according to the openflow whitepaper [6] “most new ideas go untried and untested”. Building custom hardware is difficult for several reasons : it is expensive, it is hard to keep up with the industry, and those are difficult to create and program. An over idea would be to modify devices from the market, however these are complex and closed systems. This assessment led to a new approach which is the Software-Defined Network and the Openflow Protocol which we will discuss.

3.2 Software Defined Network

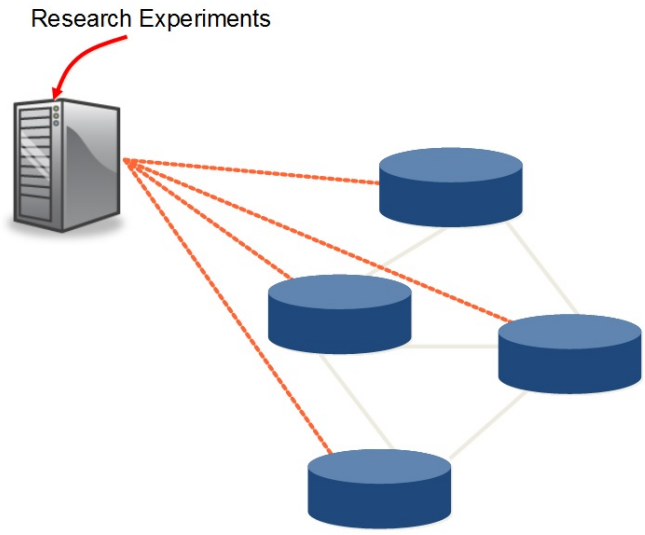

Figure 3.1 – SDN type network [7]

With SDN, a switch only has the data path functionalities, the control algorithms are moved to an external controller as showed on figure 3.1.

This approach is said to be cost-effective, it provides manageable networks, therefore network administrators have a lot more control over the network. This idea, quite simple, is exploiting the fact that on most modern switches or routers there are flow tables with enough common points so that there are multiples functionalities that can run on multiple switches from different vendors.

An Openflow switch contains some mandatory components : one or more flow tables and a group table, one or more secure channels between the switch and the controller. The Openflow Protocol is a standard protocol which can be used to communicate between the switch and the controller.

3.3 OpenFlow Protocol

The Openflow protocol is the first standard protocol for Software-Defined Network, it is maintained by the Open Networking Foundation (ONF) which is an organization which is focusing on the promotion of SDN. ONF was created in 2011 by several companies including Google, Microsoft, Yahoo!, Verizon, Facebook and Deutsche Telekom. Since that date the organization has provided multiple versions of the specification for Openflow, the latest ones (v1.5.1, v1.4.1, v1.3.5) were published in april 2015. We worked with the 1.3.5 version.

3.3.1 Flows

On a switch, a flow take entry a matching field, priority, timeouts (hard and idle), counters and an action for the matching packets, this is a rule that will define what the switch must do with the packets.

Matching field	Priority	Counters	Actions	Timeouts	Cookie	Flags

Matching field allows the flow to select the correct packet to forward among all the incoming ones.
The **Priority** is an integer, the larger it is the more important is the flow, this allow the switch to know which flow to use when there are multiple matching flows for a packet.
The **Counter** count the number of packets matched by the flow.
The **Actions** tell the switch what to do with the packet, the actions can be to forward the packet to another flow or to a group, to drop the packet, to modify the header of the packet, to forward the packet to an exit port. Some actions may be combined for example a flow can modify a packet and forward it to an exit port.
Timeouts makes sure that a flow will not remain indefinitely on the switch.The idle timeout will delete the flow if there is no packet matching the flow for a set amount of time and the hard timeout will delete the flow after a set amount of time regardless of the number of packets matching the flow.
Cookies are data added by the controller, it is not used for processing packets but may be used by the controller to filter the flows
Flags allow the controller to manage the flows.

A flow is identified by its match field and a its priority, if multiple flows exist with the same priority and matching on the same packets, matching is undefined.

3.3.2 Matching

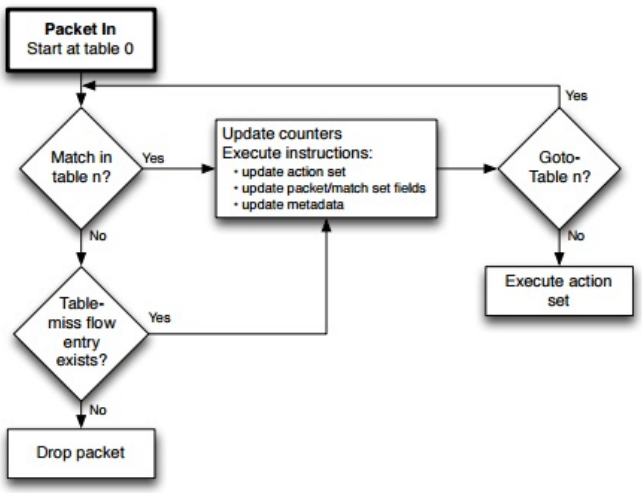

Figure 3.2 – Matching algorithm [3]

When a packet is received by the switch, as shown on figure 3.2 the switch first perform a lookup on the table 0. The switch applies the actions specified by the flow (see figure 3.3). As there may be multiple tables, if the packet is sent to another one, the switch may need to do additional lookups and repeat the process as detailled in figure 3.2.

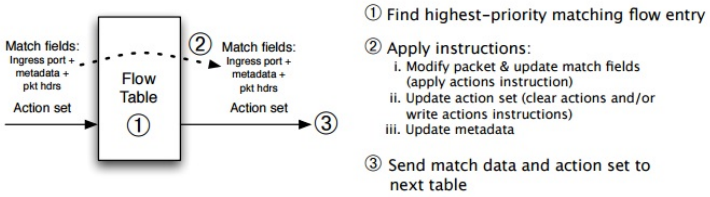

Figure 3.3 – Flow processing [4]

If they are not any flows matching the packet and there is not a table-miss flow entry, the packet is dropped. The table-miss flow entry is an entry which wildcart every packets with priority set to zero, this flow must at least support sending the packet to the controller with the proper reserved port and the reason (table-miss), and dropping the packet. This flow must be added by the controller as it is not existing by default, it behaves like any other flows.

3.3.3 Working of Openflow

There are two ways for the controller to handle the packets. It can either create the flows **proactively** or **reactively**. The first case allows an administrator to create forwarding rules using flow_mod_add messages for the packets before the switch is actually put into operation. The second case allows the controller to react to an unknown packet.

Reactive case

When an unknown packet arrives, it will either be dropped if there is not table-miss flow or it will follows the instructions set by this rule. The most common way of handling an unknown packet is to create a packetIn message. This message will contain the packet, the fact that it was triggered by a table-miss, and will be sent to the controller via the reserved port. The controller will then analyse the package, and send it back with one or multiple flow_mod_add messages stating what the switch should do with the package. After that, as long as one of the timeout is not reach, the unknown packet will be matched and processed whithout requiring any interaction with the controller.
The controller can also request the removal of a flow with an flow_mod_delete message.

Monitoring

At any given time, the controller can request the statistics for one or multiple flows with flow_stats messages.

3.4 Device

Figure 3.4 – TP link switch[5]

For our tests we are using a TP link switch (figure 3.4) and one computer. The switch is linked to the computer using four ports. Three of them are for the data path, and one is used as the reserved port for the controller. Our computer is both the controller and at the end of the datapath, this allows, using the OFtest library, to control all the test environment from one computer.
Concerning the operating system on the switch, there is an overlay : OpenVSwitch over OpenWrt. OpenWrt is a framework with package management allowing the customization of the device. OVS is "a production quality, multilayer virtual switch" (http://openvswitch.org/).

3.5 OFtest

OFtest is a framework written in Python and based on unit test, it provides a tests suite that can be used to test the openflow protocol. OFtest is part of the Floodlight project [1]. Floodlight is an open source controller and the project is sponsored by Big Switch Network (BSN). This compagny was founded in 2010 and is a pioneer in Software Defined Networks.
This framework connects to both the data and the control plane, it can therefore test and monitor both. (see figure 3.5)

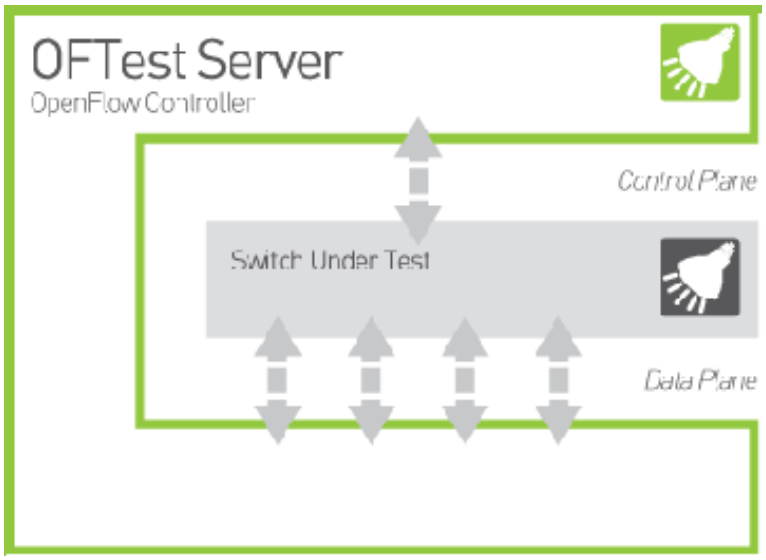

Figure 3.5 – OFtest[2]

When launching a test, the framework create a temporary floodlight controller which is used to send and receive openflow messages to the switch.

3.5.1 Structure of OFtest

The library contains both the tests and the code needed to create the environment for Openflow to work. The git repository is available at the following url : https://github.com/floodlight/oftest. The src/python folder contains the actual code for Openflow, we only modified small portions when there was some issues with the specification of Openflow (usually a wrong constant).
There are four folder for the tests : tests (openflow 1.0), tests-1.2, tests-1.3 and tests-1.4. As we were only working with the 1.3 version of openflow, the documentation provided only covers the tests-1.3 folder.
The following python files can be found in it :

- basic.py checks basic working of openflow, each functinnality can then be tested more in depth with the other files.
- actions.py checks each type of action that can be set in a flow.
- flow_mod.py checks every flow_mod message which can be used to add, remove or modify a flow.
- flow_stats.py tests that the controller can monitore the flows.

- groups.py checks that the controller can use the groups. A group is "a list of action buckets and some means of choosing one or more of those buckets to apply on a per-packet basis" (Openflow specification)
- match.y checks that every packets field can be match correctly by a flow.
- pktin_match.py checks the match sent in packet-in messages.
- role_request.py checks that the controller can switch between different roles.

There are also multiple files testing bsn features, this are only relevant for switch from bsn, we could not test them therefore those are not part of the documentation.

Note on roles

Openflow can support multiple controller, and there are three possible roles for those controllers.

- Slave : In this role a controller can not send messages to the switch that send packet or modify the state of the switch, it only has read access.
- Equal : This is the default role of the controller, in this role a controller has full access to the switch.
- Master : Same as Equal, except that there can only be one master.

3.6 Detailled problem statement

We want to create a fuzzer for the Openflow Protocol and use it on our switch, thus we need to be able to monitor and test the openflow protocol, this is what the OFtest library provide. There are hundred of tests in the OFtest library however none of them are documented. Therefore, before tempting anything on the switch we need to create the documentation to know exactly what each test is doing and why. Moreover some of those tests are malfunctioning and when it is possible we need to correct them. This work also allows us to spot some ideas that could be used for the fuzzer.

4 Documentation and corrections of OFtest

4.1 Documentation

The documentation was the first step, we started by documenting the basic tests (basic.py), and then moved on to more complicated and focuses tests. Some tests were failed by our switch, we needed to understand why so we could rectify the issue whenever possible. We designed the documentation to be as clear and complete as possible, so that a new user could use it with ease. We also wanted the user to be able to understand why we made some modifications, and to be easily capable to reverse them if necessary.
When documenting, we used Wireshark to see what messages were being exchanged between the switch and the controller, we also monitored the switch directly using the ssh connection and OVS tools.

4.1.1 Structure of the documentation

The documentation is quickly introducing Openflow and OFtest, it is not meant to be exhaustive, however this should help a newcomer to put things into perspective. The documentation provide a summary of all the test in a table with the following content :

Name of the test	Plane	Correctness	Type of error	Corrected

- The Correctness field states whether the test passed or failed by the switch before correction.
- The Type of error field states the type of error (see 4.2.2).
- The Corrected field states wheter the test was corrected or no.

When there are multiples errors, multiple types may be specified, then multiple statement are written in the corrected field (one for each error).
This is intended to provide the user with a quick way of knowing the status of a given test. Tests are in alphabetical order for each Python file, which means that for a test belonging in the action.py file, the user must go to the section related to action.py and then the tests are in alphabetical order.

The test descriptions are, in the same way, accordingly to the Python files, and them classified in alphabetical order. Each Python file in introduced by a short summary explaining what features are being tested by the file.

Each test is marked OK, or FAILED or OK (corrected) accordingly and introduced by a short summary of what it does. Then the documentation provides a chart with the different messages that are exchanged between the switch and the controller. To avoid overloading the document we did not list the messages sent by OFtest when it is creating the controller and checking the connexion as those messages are the same for each tests and do not contribute to the test itself. At the bottom of the chart there is an explanation for every unit tests conducted by the test.
Finally, there is an other table for the test which are originaly failed by the switch, this table explains (if possible) why the test is failed, and then states the correction if there is one.

The following section provides an example, the rest of the documentation can be found in the appendix.

4.1.2 basic

This test suite checks that all basic functionalites and messages of Openflow are working properly.

AsynConfigGet – OK (corrected)

This test verifies the initial async configuration.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Ask for async configuration Send async configuration
Tests		
	Check controller received an answer Check controller received async configuration reply Check controller received correct values	

Error	Correction
There is a body after the header of the request, this should not be the case and therefore this is causing an OFPT_ERROR	The body was removed (in message.py)
Tested Values were wrong according to the specification	Test was modified to match the specification

4.2 Corrections of OFtests

As stated before, a lot of tests were failed and we had to find out why and if possible to correct them.

4.2.1 Approach

We conducted the corrections alongside the documentation process, each time we encountered a test that was failing, we investigated it. The most useful tool was Wireshark as it allows the dissection of the messages, this was really helpfull in spotting values or messages that did not respected the specification of openflow.

4.2.2 Type of errors and corrections

We classified the errors into multiple types.

- Vlan
- Trailer
- Switch
- Specs
- OVS

Vlan

Those errors are caused by the switch because it is removing the Vlan tags of the packets entering it. This is caused by the fact that there is only one physical interface on the router for all ports and the router is using his own Vlan tags to differentiate the physical ports.

Trailer

Our switch is adding two bytes of data at the end of every packets it sends to the controller (VSS-Monitoring ethernet trailer). Those were not taken into account by the Oftest framework. We chose to make Oftest disregard those bytes with the option –noTrailer so that the tests could be tested.

Switch

This type regroups all errors that are related to the switch but in neither of the first two categories. For example we used this category for the tests that were failed because the switch was not handling the messages quickly enough.

Specs

Some errors are caused by some errors in OFtest regarding the specifications of Openflow, often those are errors in constants.

OVS

Some errors are caused by functionalities that are not supported by OVS (group chaining for example). Those errors were not corrected.

4.2.3 Ethernet source matching

During our tests we encountered an anormal error with the eternet source matching. This error was met when we created a flow matching on the Ethernet source field of the header of TCP packets and then tried to send some packets with the proper matching fields. Not only the packets did not match the flow and were dropped, but more importantly after this test, all packets that were sent to the switch were also drop regardless of the flowtable and the packets.
However we were not sure from where the error could come.

With that in mind, we created two virtual machines : one for a controller, and one for a virtual switch. That way we could conduct the test without the physical switch and whithout changing the network configuration of our computer. We installed OpenDayLight[1] on the virtual switch and we forged the openflow packet on the virtual controller. With this setup we were able to determine that it was the switch itself that was the cause of the failure.

[1] http://sdnhub.org/tutorials/opendaylight/

5 Fuzzing

This chapter is meant to provide an introduction to the fuzzing and some insight to what we tried with it and our setup.

5.1 Introduction to Fuzzing

Fuzzing is a technique for negative testing which means that want to verify how an application react to unexpected input or user behaviour. The goal is to find security related defects or any type of issues. Fuzzing in itself have one goal : to crash the system. The flaws discovered with this method can then be analysed and corrected. In this internship we want to create semi valid Openflow messages in large quantity, send them to the switch, and check how it handles the input. However we are leaving to the user the ability to design the tests.

5.2 Implementation

We wanted the user to be able to create his own test without having to write code by hand, that is why we created two Python files. The first one is called test_fuzzer, it is the file the user must edit to add commands. It can be found in the tests-1.3 folder.
Here is the command :
add(self,nameOfMessage,iteration,xid,length,version,type,bversion,btype,blength,bxid,data)

- nameOfMessage : Only mandatory argument, it can either be "flow_add", "echo" "flow_del" (which delete all flows regardless of the argument) or "barrier".
- iteration : number of times the message should be sent (default = 1)
- xid : if you want to modify the xid of the message (int)
- length : if you want to modify the length (int)
- version : if you want to modify the version of Openflow, (int, default = 4 for 1.3)
- type : this field is used to tell the switch what the message is, default : 14 for flow_add, 2 for echo, and 20 for barrier.
- bversion, btype, blength, bxid specified on how many bytes the field is set. It can be "!B", "!H" or "!L"

- data : this field is relevant for echo messages which can contains data, default = None.

Examples :

- add(self,"echo",xid=42,data = "this is a test")
- add(self,"flow_add", iteration = 100000, bversion="!H")

Of course the user can use the command multiple times on separate lines to create more complex tests.
The second file, that must not be modify by the user, is parser.py which is also in the tests-1.3 folder. It will create the tests and run them with the OFTest framework. At the end of the tests added by the user, there are calls to some basic tests from OFtest to check if the switch is still responding.

Those tests will send an echo request, create a flow an add it on the switch, perform a barrier and then perform two delete_all_flows requests. Unit tests from the OFTest library are called to check the responses from the switch.

To launch the tests created you must use the following command from inside the oftest folder :

./oft basic.Echo -p 6653 -H 192.168.10.2 -V 1.3 –verbose -i 1@eth11 -i 2@eth2 -i 3@eth3
Note that the port, the ip address from the controller, and the interface may change depending on your setup and your configuration.

For this fuzzer to work, some OFtest files were modify (especially those with the structure of the messages), however the library is still perfectly usable and you still can run every tests originally implemented.

5.3 Testing

Tests	Results
Large number of random messages without meaning	Switch is shutting down his socket after 7 meaningless messages, it resumes normal operations after the test.
Large number of Echo messages (small messages)	Nothing
Large number of Flow_Add messages (large messages)	Switch is shutting down his socket after 7 meaningless messages, it resumes normal operations after the test.
Large numbler of modified message. One (or multiple) Field of the message is extended (for example the id is coded on three bytes instead of two)	Messages are rejected and the switch works correctly after the test
Large number of modified message. One field is removed	Messages are rejected and the switch works correctly after the test

6 Conclusion

During this intership I was able to discover SDN and Openflow, I was not aware of those technology before. I found the reading really instructive even if at first it was quite difficult to familiarise myself with it, still I was able to learn throughout the internship everything I needed.

This internship was also the first time I coded in Python, I was not familiar with this programming language before. However the OFTest library was well written, and it was easy to understand the code even at first sight. I now feel confortable writting code in Python respecting the pep 8 style.

The fuzzing part was not really conclusive, I was unable to compromise the switch in any way as in worst case scenario the switch just stop being reachable. However the framework should be easily usable by someone else to bring new ideas and design different tests.

I felt well integrated in the team, and I was able to learn a lot from those few weeks, therefore it was a very positive experience. I still regret not having the time to conduct more tests for the fuzzing part.

bibliography / webography

[1] FloodLight. Floodlight. http://www.projectfloodlight.org/. 11

[2] FloodLight. OFtest framework. http://www.projectfloodlight.org/oftest/. 11, 21

[3] Open Networking Foundation. matching algorithm image. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf. 9, 21

[4] Open Networking Foundation. SDN type network image. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf. 9, 21

[5] TP link. TP link switch. http://www.tp-link.com/en/products/details/Archer-C7.html. 10, 21

[6] Hari Balakrishnan Guru Parulkar Larry Peterson Jennifer Rexford Scott Shenker Jonathan Turner Nick McKeown, Tom Anderson. Openflow white paper. http://archive.openflow.org/documents/openflow-wp-latest.pdf. 7

[7] OpenFlow team. SDN type network image. http://archive.openflow.org/documents/OpenFlow$_2$011.*pps*. 7, 21

List of Figures

2.1 general organisation of the LORIA (www.loria.fr/en) 3

2.2 general organisation of the inria (www.inria.fr/) 4

2.3 inria (http://www.pss-archi.eu/) 5

3.1 SDN type network [7] 7

3.2 Matching algorithm [3] 9

3.3 Flow processing [4] 9

3.4 TP link switch[5] 10

3.5 OFtest[2] 11

Résumé

Ce stage avait pour objectif de créer un fuzzer pour le protocole Openflow. Dans un premier temps il s'agissait de documenter, et de corriger les éventuels bugs, la librairie de test OFtest afin de pouvoir surveiller le switch et vérifier son état après le fuzzing. Au cours du stage il a également été question de vérifier en détail l'origine d'un problème concernant le matching sur les adresses Ethernet sources, ce problème s'est avéré être lié au matériel et non au logiciel ce qui en réduit grandement l'impact. Les résultats finaux du fuzzing ne sont pas probants, car le switch bloque systématiquement la connexion si les messages sont trop éloignés de messages Openflow ou si il y a trop d'informations et qu'il n'est pas capable de les traiter. En revanche après le test le fonctionnement du switch redevient normal et aucune anomalie n'a été détectée.

Mots-clés : OpenFlow, OFtest, Fuzzing

Abstract

This internship had for purpose to create a fuzzer for the openflow protocol. At first I had to document, and correct (if possible) the bugs, the OFTest framework so we could monitore and verify the state of the switch after the fuzzing. During the internship I also had to search in detail the origin of an issue concernint the matching on Ethernet source addresses. It turned out that this issue was due to the device and not the software thus reducing the impact. Final tests of the fuzzing were not really meaningfull as the switch is systematically blocking the connection if the messages are too different from standard Openflow mesages or if there is too much information and that it is not capable of treating it. However after the test, the switch go back to his normal behaviour, and no anomaly could be found.

Keywords : OpenFlow, OFtest, Fuzzing

A APPENDIX

OFTest

Sébastien Coorevits

June 2016

Contents

1 Openflow **5**

2 OFtest **5**

3 Setup **5**

4 VSS-Monitoring ethernet trailer **5**

5 Types of error **5**
5.1 Vlan . . . 5
5.2 Trailer . . . 5
5.3 Switch . . . 5
5.4 Specs . . . 5
5.5 OVS . . . 5

6 Usage **6**
8 6 Summary **7**

9 Openflow 1.3 **11**
9.1 basic . . . 11
9.1.1 AggregateStats – OK . . . 11
9.1.2 AsynConfigGet – OK (corrected) . . . 11
9.1.3 DefaultDrop – OK . . . 11
9.1.4 DescStats – OK . . . 12
9.1.5 Echo – OK . . . 12
9.1.6 EchoWithData – OK . . . 12
9.1.7 FeaturesRequest – OK . . . 13
9.1.8 FlowRemoveAll – OK . . . 13
9.1.9 FlowStats – OK . . . 13
9.1.10 MeterConfigStats – OK . . . 13
9.1.11 MeterFeaturesStats – OK . . . 14
9.1.12 MeterStats – OK . . . 14
9.1.13 OutputExact – OK (corrected) . . . 14
9.1.14 OutputWildcard – OK (corrected) . . . 15
9.1.15 PacketInExact – OK (corrected) . . . 15
9.1.16 PacketInMiss – OK (corrected) . . . 16
9.1.17 PacketInWildcard – OK (corrected) . . . 16
9.1.18 PacketOut – OK (corrected) . . . 16
9.1.19 PortConfigMod – OK . . . 17
9.1.20 PortDescStats – OK . . . 17
9.1.21 PortStats – OK . . . 17
9.1.22 GroupDescStats – OK . . . 18
9.1.23 GroupFeaturesStats – OK . . . 18
9.1.24 GroupStats – OK . . . 18
9.1.25 QueueStats – OK . . . 18
9.1.26 TableFeaturesStats – DISABLED . . . 18
9.1.27 TableStats – OK . . . 19
9.2 actions . . . 19
9.2.1 DecIpv4TTL – OK . . . 19
9.2.2 DecIpv6HopLimit – OK . . . 19
9.2.3 Output – OK . . . 19
9.2.4 OutputMultiple – OK . . . 19
9.2.5 PopVlan – FAILED . . . 20
9.2.6 PushVlan – OK . . . 20
9.2.7 PushVlanPcp – OK . . . 20
9.2.8 PushVlanVid – FAILED . . . 20
9.2.9 PushVlanVidPcp – FAILED . . . 21
9.2.10 SetEthDst – OK . . . 21
9.2.11 SetEthSrc – OK . . . 21
9.2.12 SetIPv4Dst – OK . . . 21
9.2.13 SetIPv4Src – OK . . . 22
9.2.14 SetIPv6Dst – OK . . . 22
9.2.15 SetIPv6Flabel – OK . . . 22
9.2.16 SetIPv6Flabel_NonZeroDSCPandECN – OK . . . 22
9.2.17 SetIPv6Src – OK . . . 22
9.2.18 SetIpv4DSCP_NonZeroECN – OK . . . 23
9.2.19 SetIpv4Dscp – OK . . . 23
9.2.20 SetIpv4ECN – OK . . . 23
9.2.21 SetIpv4ECN_NonZeroDSCP – OK . . . 23
9.2.22 SetIpv4TTL – OK . . . 23
9.2.23 SetIpv6DSCP_NonZeroECNandFlabel – OK . . . 24
9.2.24 SetIpv6Dscp – OK . . . 24
9.2.25 SetIpv6ECN – OK . . . 24
9.2.26 SetIpv6ECN_NonZeroDSCPandFlabel – OK . . . 24
9.2.27 SetIpv6HopLimit – OK . . . 24
9.2.28 SetTCPDst – OK . . . 25
9.2.29 SetTCPDsrc – OK . . . 25

9.2.30 SetUDPDst – OK . . . 25
9.2.31 SetUDPDsrc – OK . . . 25
9.2.32 SetVlanPcp – FAILED . . . 25
9.2.33 SetVlanVid – FAILED . . . 26
9.3 flow_mod . . . 26
9.3.1 Overwrite – OK . . . 26
9.4 flow_stats . . . 26
9.4.1 AllFlowStats – FAILED . . . 26
9.4.2 CookieFlowStats – OK . . . 28
9.5 groups . . . 28
9.5.1 GroupAdd – OK (corrected) . . . 28
9.5.2 GroupAddAllWeight – FAILED . . . 28
9.5.3 GroupAddExisting – OK (corrected) . . . 29
9.5.4 GroupAddIndirectBuckets – OK . . . 29
9.5.5 GroupAddIndirectWeight – FAILED . . . 29
9.5.6 GroupAddInvalidAction – FAILED . . . 30
9.5.7 GroupAddInvalidID – OK (corrected) . . . 31
9.5.8 GroupAddMaxID – OK (corrected) . . . 31
9.5.9 GroupAddMinimumInvalidID – OK (corrected) . . . 32
9.5.10 GroupAddSelectNoWeight – FAILED . . . 32
9.5.11 GroupDeleteAll – OK (corrected) . . . 32
9.5.12 GroupDeleteExisting – OK (corrected) . . . 33
9.5.13 GroupDeleteNonexisting – OK . . . 33
9.5.14 GroupDescStats – OK (corrected) . . . 33
9.5.15 GroupFlowSelect – OK . . . 34
9.5.16 GroupModify – OK (corrected) . . . 34
9.5.17 GroupModifyEmpty – OK (corrected) . . . 35
9.5.18 GroupModifyInvalidID – OK (corrected) . . . 35
9.5.19 GroupModifyLoop – OK (corrected) . . . 36
9.5.20 GroupModifyNonexisting – FAILED . . . 36
9.5.21 GroupStats – OK (corrected) . . . 37
9.5.22 GroupStatsAll – OK (corrected) . . . 37
9.5.23 GroupStatsNonExistent – OK . . . 38
9.5.24 SelectFwdEmpty – OK (corrected) . . . 38
9.5.25 SelectFwdSingle – OK (corrected) . . . 38
9.5.26 SelectFwdSpread – OK (corrected) . . . 39
9.6 match . . . 39
9.6.1 ArpOp – OK . . . 39
9.6.2 ArpSPA – OK . . . 40
9.6.3 ArpSPAMasked – OK . . . 40
9.6.4 ArpSPASubnetMasked – OK . . . 41
9.6.5 ArpTPA – OK . . . 41
9.6.6 ArpTPAMasked – OK . . . 42
9.6.7 ArpTPASubnetMasked – OK . . . 42
9.6.8 EthDst – OK . . . 43
9.6.9 EthDstBroadcast – OK . . . 43
9.6.10 EthDstMasked – OK . . . 44
9.6.11 EthDstMulticast – OK . . . 44
9.6.12 EthSrc – FAILED . . . 45
9.6.13 EthSrcBroadcast – FAILED . . . 45
9.6.14 EthSrcMasked – FAILED . . . 46
9.6.15 EthSrcMulticast – FAILED . . . 46
9.6.16 EthTypeARP – OK . . . 47
9.6.17 EthTypeIPv4 – OK (corrected) . . . 47
9.6.18 EthTypeIPv6 – OK (corrected) . . . 48
9.6.19 EthTypeNone – OK . . . 49
9.6.20 IPv4Dscp – OK . . . 49
9.6.21 IPv4Dst – OK . . . 49
9.6.22 IPv4DstMasked – OK . . . 51
9.6.23 IPv4DstSubnetMasked – OK . . . 51
9.6.24 IPv4Ecn – OK . . . 52
9.6.25 IPv4ICMPCode – OK . . . 52
9.6.26 IPv4ICMPType – OK . . . 53
9.6.27 IPv4ProtoICMP – OK . . . 53
9.6.28 IPv4ProtoTCP – OK . . . 54
9.6.29 IPv4ProtoUDP – OK . . . 54
9.6.30 IPv4Src – OK . . . 55
9.6.31 IPv4SrcMasked – OK . . . 55
9.6.32 IPv4SrcSubnetMasked – OK . . . 56
9.6.33 IPv4TCPDst – OK . . . 56
9.6.34 IPv4TCPDstMasked – OK . . . 56
9.6.35 IPv4TCPSrc – OK . . . 57
9.6.36 IPv4TCPSrcMasked – OK . . . 58
9.6.37 IPv4UDPDst – OK . . . 58
9.6.38 IPv4UDPDstMasked – OK . . . 59
9.6.39 IPv4UDPSrc – OK . . . 59
9.6.40 IPv4UDPSrcMasked – OK . . . 60
9.6.41 IPv6Dscp – OK . . . 61
9.6.42 IPv6Dst – OK . . . 61

9.6.43 IPv6DstMasked – OK . . . 62
9.6.44 IPv6DstSubnetMasked – OK . . . 63
9.6.45 IPv6Ecn – OK . . . 63
9.6.46 IPv6ICMPCode – OK . . . 64
9.6.47 IPv6ICMPType – OK . . . 65
9.6.48 IPv6ProtoICMP – OK . . . 65
9.6.49 IPv6ProtoTCP – OK . . . 66
9.6.50 IPv6ProtoUDP – OK . . . 66
9.6.51 IPv6Src – OK . . . 67
9.6.52 IPv6SrcMasked – OK . . . 67
9.6.53 IPv6SrcSubnetMasked – OK . . . 68
9.6.54 IPv6TCPDst – OK . . . 68
9.6.55 IPv6TCPSrc – OK . . . 69
9.6.56 IPv6UDPDst – OK . . . 69
9.6.57 IPv6UDPSrc – OK . . . 70
9.6.58 InPort – OK . . . 70
9.6.59 VlanAbsent – FAILED . . . 71
9.6.60 VlanExact – FAILED . . . 72
9.6.61 VlanPCP – FAILED . . . 73
9.6.62 VlanPCPAnyVID – FAILED . . . 73
9.6.63 VlanPCPMasked – FAILED . . . 74
9.6.64 VlanPresent – FAILED . . . 75
9.6.65 VlanVID – FAILED . . . 75
9.6.66 VlanVIDMasked – FAILED . . . 76
9.7 role_request . . . 77
9.7.1 RolePermissions – OK . . . 77
9.7.2 RoleRequestEqualToMaster – OK (corrected) . . . 78
9.7.3 RoleRequestEqualToSlave – OK (corrected) . . . 79
9.7.4 RoleRequestNoChange – OK . . . 80
9.7.5 RoleRequestSlaveToMaster – OK . . . 81
9.7.6 SlaveNoPacketIn – OK . . . 82
9.8 pktin_match . . . 83
9.8.1 VlanAbsent – OK . . . 83
9.8.2 VlanVid – FAILED . . . 83

10 Bsn messages **83**

1 Openflow

While traditional routing algorithms are distributed, meaning that each switch handles his routing tables on his own.
SDN introduces the idea of decoupling the planes (data and control planes) which makes it easier to test protocols, adds more flexibility, more performance, and allows central management. This means that the switchs are only forwarding devices and that there is a controller that handles the control parts.
Openflow is a communication protocol based on TCP for the communication between the controller and the switchs. The Openflow protocol defines the control messages which are used to control the flow tables.

2 OFtest

OFtest is a test suite that test the compliance of a setup with the openflow specification. With this framework we can test an openflow switch with only one computer witch is at the end of both the data planes and the control plane.
This document intend to provide some documentation to the framework and to provide some explanation about some test which were not passed by our switch. We brought some corrections to the OFtest framework based on our setup, their are detailed in this document with the explanation.

3 Setup

For our tests we used a TP-link router with OpenWrt and Open vSwitch and a computer under linuxMint. We have three ports on the router that are connected to the computer as data ports and one that is connected to the computer as the control port. There is only one physical interface on the router for all three ports and the router is using his own Vlan tags to differentiate the physical ports.

4 VSS-Monitoring ethernet trailer

Our switch is adding two bytes of data at the end of every packets it sends to the controller. Those were not taken into account by the Oftest framework. We chose to make Oftest disregard those bytes with the option –noTrailer.

5 Types of error

5.1 Vlan

Those errors are caused by the switch because it is removing the Vlan tags of the packets entering it.

5.2 Trailer

Those errors are caused by the trailer added by the switch.

5.3 Switch

This type regroups all errors that are related to the switch but in neither of the first two categories.

5.4 Specs

Some errors are caused by some errors in OFtest regarding the specifications of Openflow, often those are errors in constants.

5.5 OVS

Some errors are caused by functionalities that are not supported by OVS (group chaining for example).

6 Usage

To use the library :
./oft file.test -p 6653 -H IP address_to_listen_on -V 1.3 –verbose -i ofport1@interface1 -i ofport2@interface2 -i ofport3@interface3
Example :

- ./oft basic -p 6653 -H 192.168.10.2 -V 1.3 –verbose -i 1@eth11 -i 2@eth2 -i 3@eth3
- ./oft test_fuzzer -p 6653 -H 192.168.10.2 -V 1.3 –verbose -i 1@eth11 -i 2@eth2 -i 3@eth3

We have introduced two new options : –correction and –noTrailer.
The –correction option allows the user to apply all corrections we provide that are not related to the switch. The –noTrailer option allows the OFtest library to disregard the two last bytes of all packets coming from the switch.

7 Pré-requisite [1]

You will need the following software to run OFtest :

- Python 2.7 (sudo yum install python or sudo apt-get install python)
- Scapy (sudo yum install scapy or sudo apt-get install scapy)

You need root access to run oft. You need an Openflow switch instance actively attempting to connect to a controller on your computer. If the switch is trying to connect to the computer via a port which is not 6653, do not forget to change that in the ./oft command.

[1] https://github.com/floodlight/oftest/blob/master/README.md

8 Summary

Test name	plane	Correctness	type of error	corrected
basic				
AggregateStats	Control	OK	None	No
AsynConfigGet	Control	Failed	Specs	Yes
DefaultDrop	Control	Ok	None	No
DesStats	Control	Ok	None	No
Echo	Control	Ok	None	No
EchoWithData	Control	Ok	None	No
FeaturesRequest	Control	Ok	None	No
FlowRemoveAll	Control	Ok	None	No
FlowStats	Control	Ok	None	No
MeterConigStats	Control	Ok	None	No
MeterStats	Control	Ok	None	No
OutputExact	Data	Failed	Trailer, Switch	Yes
OutputWildcard	Data	Failed	Trailer, Switch	Yes
PacketInExact	Control	Ok	None	No
PacketInMiss	Control	Ok	None	No
PacketInWildcard	Control	Ok	None	No
PacketOut	Control	Failed	Trailer	Yes
PortConfigMod	Control	Ok	None	No
PortDescStats	Control	Ok	None	No
PortStats	Control	Ok	None	No
GroupDescStats	Control	Ok	None	No
* GroupFeaturesStats	Control	Ok	None	No
GroupStats	Control	Ok	None	No
QueueStats	Control	Ok	None	No
TableFeaturesStats	None	Disabled	None	No
TableStats	Control	Ok	None	No
actions				
DecIpv4TTL	Control	Failed	Trailer	Yes
DecIpv6HopLimit	Control	Failed	Trailer	Yes
Output	Control	Failed	Trailer	Yes
OutputMultiple	Control	Failed	Trailer	Yes
PopVlan	Control	Failed	Trailer, Vlan	Yes, No
PushVlanPcp	Control	Failed	Trailer	Yes
PushVlanVip	Control	Failed	Trailer, Vlan	Yes, No
PushVlanVipPcp	Control	Failed	Trailer, Vlan	Yes, No
SetEthDst	Control	Failed	Trailer	Yes
SetEthSrc	Control	Failed	Trailer	Yes
SetIPv4Dst	Control	Failed	Trailer	Yes
SetIPv4Src	Control	Failed	Trailer	Yes
SetIPv6Dst	Control	Failed	Trailer	Yes
SetIPv6Flabel	Control	Failed	Trailer	Yes
SetIPv6Flabel_NonZeroDSCPandECN	Control	Failed	Trailer	Yes
SetIPv6Src	Control	Failed	Trailer	Yes
SetIPv4DSCP_NonZeroECN	Control	Failed	Trailer	Yes
SetIPv4Dscp	Control	Failed	Trailer	Yes
SetIPv4ECN	Control	Failed	Trailer	Yes
SetIPv4ECN_NonZeroDSCP	Control	Failed	Trailer	Yes
SetIPv4TTL	Control	Failed	Trailer	Yes
SetIPv6DSCP_NonZeroECNandFlabel	Control	Failed	Trailer	Yes
SetIPv6DSCP	Control	Failed	Trailer	Yes
SetIPv6ECN	Control	Failed	Trailer	Yes
SetIPv6ECN_NonZeroDSCPandFlabel	Control	Failed	Trailer	Yes
SetIPv6HopLimit	Control	Failed	Trailer	Yes

SetTCPDst	Control	Failed	Trailer	Yes
SetTCPSrc	Control	Failed	Trailer	Yes
SetUDPDst	Control	Failed	Trailer	Yes
SetUDPSrc	Control	Failed	Trailer	Yes
SetVlanPcp	Control	Failed	Trailer, Vlan	Yes, No
SetVlanVid	Control	Failed	Trailer, Vlan	Yes, No
Flow mod				
Overwrite	Control	Ok	None	No
Flow stats				
AllFlowStats	Control	Failed	?	No
CookieFlowStats	Control	Ok	None	No
groups				
GroupAdd	Control	Failed	Specs	Yes
GroupAddAllWeight	Control	Failed	OVS	No
GroupAddExisting	Control	Failed	Specs	Yes
GroupAddIndirectBuckets	Control	Ok	None	No
GroupAddIndirectWeight	Control	Failed	Specs, OVS	Yes, No
GroupAddInvalidID	Control	Failed	Specs, OVS	Yes, No
GroupAddInvalidAction	Control	Failed	Specs, OVS	Yes, No
GroupAddMaxID	Control	Failed	Specs	Yes
GroupAddMinimumInvalidID	Control	Failed	Specs	Yes
GroupAddSelectNoWeight	Control	Failed	OVS	No
GroupDeleteAll	Control	Failed	Specs	Yes
GroupDeleteExisting	Control	Failed	Specs	Yes
GroupDeleteNonExisting	Control	Ok	None	No
GroupDescStats	Control	Failed	Specs	Yes
GroupFlowSelect	Control	Ok	None	No
GroupModify	Control	Failed	Specs	Yes
GroupModifyEmpty	Control	Failed	Specs	Yes
GroupModifyInvalidID	Control	Failed	Specs	Yes
GroupModifyLoop	Control	Failed	Specs, OVS	Yes, No
GroupModifyNonExisting	Control	Failed	Specs	Yes
GroupStats	Control	Failed	Specs	Yes
GroupStatsAll	Control	Failed	Specs	Yes
GroupStatsNonExistent	Control	Ok	None	No
SelectFwdEmpty	Control	Failed	Trailer	Yes
SelectFwdSingle	Control	Failed	Specs	Yes
SelectFwdSpread	Control	Failed	Switch, Specs	Yes, Yes
match				
ArpOP	Data	Failed	Trailer	Yes
ARpSPA	Data	Failed	Trailer	Yes
ArpSPAMasked	Data	Failed	Trailer	Yes
ArpSPASubnetMasked	Data	Failed	Trailer	Yes
ArpTPA	Data	Failed	Trailer	Yes
ArpTPAMasked	Data	Failed	Trailer	Yes
ArpTPASubnetMasked	Data	Failed	Trailer	Yes
EthDst	Data	Failed	Trailer	Yes
EthDstBroadcast	Data	Failed	Trailer	Yes
EthDstMulticast	Data	Failed	Trailer	Yes
EthSrc	Data	Failed	Trailer, Switch	Yes, No
EthSrcBroadCast	Data	Failed	Trailer, Switch	Yes, No
EthSrcMasked	Data	Failed	Trailer, Switch	Yes, No
EthSrcMulticast	Data	Failed	Trailer, Switch	Yes, No
EthTypeARP	Data	Failed	Trailer	Yes
EthTypeIPv4	Data	Failed	Trailer, Vlan	Yes, Yes
EthTypeIPv6	Data	Failed	Trailer, Vlan	Yes, Yes

EthTypeNone	Data	Ok	None	Yes
IPv4Dscp	Data	Failed	Trailer	Yes
IPv4Dst	Data	Failed	Trailer	Yes
IPv4DstMasked	Data	Failed	Trailer	Yes
IPv4DstSubnetMasked	Data	Failed	Trailer	Yes
IPv4Ecn	Data	Failed	Trailer	Yes
IPv4ICMPCode	Data	Failed	Trailer	Yes
IPv4ICMPType	Data	Failed	Trailer	Yes
IPV4ProtoICMP	Data	Failed	Trailer	Yes
IPv4ProtoTCP	Data	Failed	Trailer	Yes
IPv4ProtoUDP	Data	Failed	Trailer	Yes
IPv4Src	Data	Failed	Trailer	Yes
IPv4SrcMasked	Data	Failed	Trailer	Yes
IPv4SrcSubnetMasked	Data	Failed	Trailer	Yes
IPv4TCPDst	Data	Failed	Trailer	Yes
IPv4TCPDstMasked	Data	Failed	Trailer	Yes
IPv4TCPSrc	Data	Failed	Trailer	Yes
IPv4TCPSrcMasked	Data	Failed	Trailer	Yes
IPv4UDPDst	Data	Failed	Trailer	Yes
IPv4UDPDstMasked	Data	Failed	Trailer	Yes
IPv4UDPSrc	Data	Failed	Trailer	Yes
IPv4UDPSrcMasked	Data	Failed	Trailer	Yes
IPv6Dscp	Data	Failed	Trailer	Yes
IPv6Dst	Data	Failed	Trailer	Yes
IPv6DstMasked	Data	Failed	Trailer	Yes
IPv6DstSubnetMasked	Data	Failed	Trailer	Yes
IPv6Ecn	Data	Failed	Trailer	Yes
IPv6ICMPCode	Data	Failed	Trailer	Yes
IPv6ICMPType	Data	Failed	Trailer	Yes
IPv6ProtoICMP	Data	Failed	Trailer	Yes
IPv6ProtoTCP	Data	Failed	Trailer	Yes
IPv6ProtoUDP	Data	Failed	Trailer	Yes
IPv6Src	Data	Failed	Trailer	Yes
IPv6SrcMasked	Data	Failed	Trailer	Yes
IPv6SrcSubnetMasked	Data	Failed	Trailer	Yes
IPv6TCPDst	Data	Failed	Trailer	Yes
IPv6TCPSrc	Data	Failed	Trailer	Yes
IPv6UDPDst	Data	Failed	Trailer	Yes
IPv6UDPSrc	Data	Failed	Trailer	Yes
InPort	Data	Failed	Trailer	Yes
VlanAbsent	Data	Failed	Trailer, Vlan	Yes, No
VlanExact	Data	Failed	Trailer, Vlan	Yes, No
VlanPCP	Data	Failed	Trailer, Vlan	Yes, No
VlanPCPAnyVID	Data	Failed	Trailer, Vlan	Yes, No
VlanPCPMasked	Data	Failed	Trailer, Vlan	Yes, No
VlanPresent	Data	Failed	Trailer, Vlan	Yes, No
VlanVID	Data	Failed	Trailer, Vlan	Yes, No
VlanVIDMasked	Data	Failed	Trailer, Vlan	Yes, No
role_request				
RolePermissions	Control	OK	None	No
RoleRequestEqualToMaster	Control	Failed	Specs(Switch)	Intended behaviour
RoleRequestEqualToSlave	Control	Failed	Specs(Switch)	Intended behaviour
RoleRequestNoChange	Control	Ok	None	No
RoleRequestSlaveToMaster	Control	Ok	None	No
SlaveNoPacketIn	Control	Ok	None	No
pktin_match				
VlanAbsent	Data	Ok	None	No

VlanVid	Data	Failed	Vlan	No

9 Openflow 1.3

9.1 basic

This test suite checks that all basic functionalites and messages of Openflow are working properly.

9.1.1 AggregateStats – OK

This test checks that the Aggregate flow stats multipart transaction is working.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	MultipartRequest OFPMP_AGGREGATE MultipartReply OFPMP_AGGREGATE
Tests		
	Check controller received an answer Check controller received only one answer (flag==0)	

9.1.2 AsynConfigGet – OK (corrected)

This test verifies the initial async configuration.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Ask for async configuration Send async configuration
Tests		
	Check controller received an answer Check controller received async configuration reply Check controller received correct values	

Error	Correction
There is a body after the header of the request, this should not be the case and therefore this is causing an OFPT_ERROR	The body was removed (in message.py)
Tested Values were wrong according to the specification	Test was modified to match the specification

9.1.3 DefaultDrop – OK

This test checks that with an empty flow table, any packets are correctly dropped.

Pkt1 Pkt2 Pkt3	Controller ⇒ Switch Controller ⇒ Switch Dataplane ⇒ Switch	Flow_Mod : OFPC_DELETE Barrier Tcp packet
Tests		
	Check there is not any packetIn messages Check there is not any packet on the dataplanes	

9.1.4 DescStats – OK

This test checks that the controller can request the switch description.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_DESC = 0 Send description (Manufacturer desc., Hardware Desc., Software desc., Serial no., Datapath desc.)
Tests		
	Check for any answer Check flags = 0 (oly one answer)	

9.1.5 Echo – OK

This test checks that the switch replies correctly at an echo request with no Data.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Send an echo request Send an echo reply
Tests		
	Check controller received an answer Check that the answer is an echo reply Check that the answer and the request have the same id Check that the answer's data is empty	

9.1.6 EchoWithData – OK

This test checks that the switch replies correctly at an echo request with Data.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Send an echo request Send an echo reply
Tests		
	Check controller received an answer Check that the answer is an echo reply Check that the answer and the request have the same id Check that the answer's data is the same as the request's	

9.1.7 FeaturesRequest – OK

This test checks that the switch replies correctly at a feature request (feature requests allow the controller to request the capabilities of the switch)

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Send a feature request Send a feature reply
Tests		
	Check controller received an answer Check that the answer is a feature reply Check that the answer and the request have the same id	

9.1.8 FlowRemoveAll – OK

This test checks that if we add multiples new flows and then delete them, none are left.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Send a feature request Send a feature reply
Tests		
	Check controller received an answer Check that the answer is a feature reply Check that the answer and the request have the same id	

9.1.9 FlowStats – OK

This test checks that the switch replies to a multipart request for information about individual flow entries.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_FLOW = 1. Send information for matching flow (length of entry, table_id, duration, duration_nsec, priority, idle_timeout, hard_timeout, flags, cookie, packet_count, byte_count, matching information)
Tests		
	Check controller received an answer	

9.1.10 MeterConfigStats – OK

This test checks that the switch replies to a multipart request for configuration for all meters.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_METER_CONFIG = 10. Send information for matching meters (length of entry, flags, meter id)
Tests		
	Check controller received an answer	

9.1.11 MeterFeaturesStats – OK

This test checks that the switch replies to a multipart request for the set of features of the metering system.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_METER_FEATURES = 11. Send information for metering system (maximum number of meters, bitmaps of OFPMBT_* values supported, Bitmaps of "ofp_meter_flags", Maximum bands per meters, Maximum color value)
Tests		
	Check controller received an answer	

9.1.12 MeterStats – OK

This test checks that the switch replies to a multipart request for the statistics of all meters.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_METER_FEATURES = 11. Send information for metering system (meter id, length of entry, number of flows bound to meter, number of packets in input, number of bytes in input)
Tests		
	Check controller received an answer	

9.1.13 OutputExact – OK (corrected)

This test checks that when the controller adds a flow directing matching packets to a specific port A, sending a packet to any other port results in an output to A.

Pkt1 Pkt2	Controller ⇒ Switch Controller ⇒ Switch	Flow_Mod : OFPC_DELETE. Barrier.
Loop on all ports		
Pkt(i) Pkt(i+1) Pkt(i+2)	Controller ⇒ Switch Dataplane ⇒ Switch Switch ⇒ Dataplane	Flow_Mod : OFPC_ADD. tcp packet. tcp packet.
Tests		
	Check that for each portA, sending a packet to any other port results in an output to A	

Error	Correction
The switch was not processing each flow add message quick enough and therefore packets were sent and expected to match a rule which was not yet available	A 1 second delay was added

9.1.14 OutputWildcard – OK (corrected)

This test checks that when the controller adds a flow directing all packets to a specific port A, sending a packet to any other port results in an output to A.

Pkt1 Pkt2	Controller ⇒ Switch Controller ⇒ Switch	Flow_Mod : OFPC_DELETE. Barrier.
Loop on all ports		
Pkt(i) Pkt(i+1) Pkt(i+2)	Controller ⇒ Switch Dataplane ⇒ Switch Switch ⇒ Dataplane	Flow_Mod : OFPC_ADD. tcp packet. tcp packet.
Tests		
	Check that for each port A, sending a packet to any other port results in an output to A	

Error	Correction
The switch was not processing each flow add message quick enough and therefore packets were sent and expected to match a rule which was not yet available	A 1 second delay was added

9.1.15 PacketInExact – OK (corrected)

This test checks that when a flow redirecting matching packets to the controller is added, sending a packet to a dataplane port correctly sends the packet_in to the controller with the reason being the redirection.

Pkt1 Pkt2 Pkt3	Controller ⇒ Switch Controller ⇒ Switch Controller ⇒ Switch	Flow_Mod : OFPC_DELETE. Barrier. Flow_Mod : OFPC_ADD (redirection toward controller).
Loop on all ports		
Pkt4	Dataplanes ⇒ Switch	tcp packet
Tests		
	Check controller received a packet_in	

Error	Correction
The switch is adding two bits at the end of each packets it sends (vss monitoring trailer), which were not planned by OFtest	The two last bits of each packet sent by the switch were ignored

9.1.16 PacketInMiss – OK (corrected)

This test checks that when a flow redirecting all packets to the controller but with priority set to 0 is added, sending a packet to a dataplane port correctly sends a packet_in to the controller with the reason being a table_miss.

Pkt1 Pkt2 Pkt3	Controller ⇒ Switch Controller ⇒ Switch Controller ⇒ Switch	Flow_Mod : OFPC_DELETE. Barrier. Flow_Mod : OFPC_ADD (priority set to 0).
Loop on all ports		
Pkt4	Dataplanes ⇒ Switch	tcp packet
Tests		
	Check controller received a packet_in	

Error	Correction
The switch is adding two bits at the end of each packets it sends (vss monitoring trailer), which were not planned by OFtest	The two last bits of each packet sent by the switch were ignored

9.1.17 PacketInWildcard – OK (corrected)

This test checks that when a flow redirecting all packets to the controller is added, sending a packet to a dataplane port correctly sends a packet_in to the controller with the reason being the redirection.

Pkt1 Pkt2 Pkt3	Controller ⇒ Switch Controller ⇒ Switch Controller ⇒ Switch	Flow_Mod : OFPC_DELETE. Barrier. Flow_Mod : OFPC_ADD (matching all packets, redirection towar controller).
Loop on all ports		
Pkt4	Dataplanes ⇒ Switch	tcp packet
Tests		
	Check controller received a packet_in	

Error	Correction
The switch is adding two bits at the end of each packets it sends (vss monitoring trailer), which were not planned by OFtest	The two last bits of each packet sent by the switch were ignored

9.1.18 PacketOut – OK (corrected)

This test checks that when the controller sends a packetOut a packet is send on the appropriate port.

Loop on all ports		
Pkt1	Controller ⇒ Switch	Packet_out.
Loop on all ports		
Pkt4	Switch ⇒ Dataplanes	tcp packet
Tests		
	Check Dataplanes received a packet_out	

Error	Correction
The switch is adding two bits at the end of each packets it sends (vss monitoring trailer), which were not planned by OFtest	The two last bits of each packet sent by the switch were ignored

9.1.19 PortConfigMod – OK

This test checks that the controller can request the configuration of a port, change it, verified it is changed and then set it back.

Pkt1	Controller ⇒ Switch	Multipart request with OFPMP_PORT_DESC = 13
Pkt2	Switch ⇒ Controller	Send port's configuration
Pkt3	Controller ⇒ Switch	OFPT_PORT_MOD : change a bit in the configuration
Pkt4	Controller ⇒ Switch	Multipart request with OFPMP_PORT_DESC = 13
Pkt5	Switch ⇒ Controller	Send port's configuration
Pkt6	Controller ⇒ Switch	OFPT_PORT_MOD : change back the bit in the configuration
Tests		
	Check if controller received port configuration Check there is no issue with the modification message Check if controller received modified port configuration Check that the modification was made Check there is no issue with the second modification message	

9.1.20 PortDescStats – OK

This test checks that the switch replies to a multipart request for the description of all ports.

Pkt1	Controller ⇒ Switch	Multipart request with OFPMP_PORT_DESC = 13.
Pkt2	Switch ⇒ Controller	Multipart reply with the configuration (config, state, features).
Tests		
	Check controller received an answer	

9.1.21 PortStats – OK

This test checks that the switch replies to a multipart request for the statistics of all ports.

Pkt1	Controller ⇒ Switch	Multipart request with OFPMP_PORT_STATS = 4.
Pkt2	Switch ⇒ Controller	Multipart reply with the statistics (no of port, number of : received packets, transmitted packets, received bytes, transmitted bytes, packets dropped by RX, packets dropped by TX, received errors, transmitted errors, frame alignment errors, packets with RX overrun, CRC errors, collisions; duration_sec, duraction_nsec).
Tests		
	Check controller received an answer	

9.1.22 GroupDescStats – OK

This test checks that the switch replies to a multipart request for the description of all groups.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_GROUP_DESC = 7. Multipart reply with the description (type, group_id).
Tests		
	Check controller received an answer	

9.1.23 GroupFeaturesStats – OK

This test checks that the switch replies to a multipart request for the features of all groups.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_GROUP_DESC = 8. Multipart reply with the features (types supported, capabilities supported, maximum number of groups for each type, actions supported).
Tests		
	Check controller received an answer	

9.1.24 GroupStats – OK

This test checks that the switch replies to a multipart request for the statistics of all groups.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_GROUP = 6. Multipart reply with the statistics (group_id, number of flows or groups that directly forward to this group, packet_count, byte_count, duraction_sec, duraction_nsec).
Tests		
	Check controller received an answer	

9.1.25 QueueStats – OK

This test checks that the switch replies to a multipart request for the statistics of all queues.

Pkt1 Pkt2	Controller ⇒ Switch Switch ⇒ Controller	Multipart request with OFPMP_QUEUE = 5. Multipart reply with the statistics (queue_id, number of transmitted bytes, number of transmitted packets, number of packets dropped due to overrun, duraction_sec, duraction_nsec).
Tests		
	Check controller received an answer	

9.1.26 TableFeaturesStats – DISABLED

Tablefeatures are not yet supported.

9.1.27 TableStats – OK

This test checks that the switch replies to a multipart request for the statistics of all queues.

Pkt1	Controller ⇒ Switch	Multipart request with OFPMP_TABLE= 7.
Pkt2	Switch ⇒ Controller	Multipart reply with the statistics (table_id, number of : actives entries, packets looked up in the table, packets that hit the table).
Tests		
	Check controller received an answer	

9.2 actions

This test suite checks that the controller can add all the different types of action to a flow, and that those action works properly.

9.2.1 DecIpv4TTL – OK

This test checks that a flow is correctly able to decrement the Ipv4 TTL field.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with action dec_ttl).
Pkt2	Dataplane ⇒ Switch	tcp packet with ttl = 10
Pkt3	Switch ⇒ Dataplane	tcp packet with ttl = 9
Tests		
	Check a packet came out	

9.2.2 DecIpv6HopLimit – OK

This test checks that a flow is correctly able to decrement the Ipv6 hop limit.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with action dec_ttl).
Pkt2	Dataplane ⇒ Switch	tcp packet with hlim = 10
Pkt3	Switch ⇒ Dataplane	tcp packet with hlim = 9
Tests		
	Check a packet came out	

9.2.3 Output – OK

This test checks that a flow is correctly able to forward a packet to a single port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check a packet came out	

9.2.4 OutputMultiple – OK

This test checks that a flow is correctly able to forward a packet to three ports.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with three forwards rules)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check a packet came out	

9.2.5 PopVlan – FAILED

This test checks that a flow is correctly able to pop the vlan from a packet

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the pop vlan option)
Pkt2	Dataplane ⇒ Switch	tcp packet (with vlan)
Pkt3	Switch ⇒ Dataplane	tcp packet (with vlan)
Tests		
	Check if a packet came out	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.2.6 PushVlan – OK

This test checks that a flow is correctly able to push a vlan in the header of a packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the push vlan instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet (without vlan)
Pkt3	Switch ⇒ Dataplane	tcp packet (with vlan)
Tests		
	Check if a packet came out	

9.2.7 PushVlanPcp – OK

This test checks that a flow is correctly able to push a vlan in the header of a packet and to set the priority of the vlan (pcp).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the push vlan and set pcp instructions)
Pkt2	Dataplane ⇒ Switch	tcp packet (without vlan)
Pkt3	Switch ⇒ Dataplane	tcp packet (with vlan)
Tests		
	Check if a packet came out	

9.2.8 PushVlanVid – FAILED

This test checks that a flow is correctly able to push a vlan in the header of a packet and to set the id of the vlan.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the push vlan and set vid instructions)
Pkt2	Dataplane ⇒ Switch	tcp packet (without vlan)
Pkt3	Switch ⇒ Dataplane	tcp packet (with vlan)
Tests		
	Check if a packet came out	

Error	Correction
The flow mod fails and the switch returns an OFPBAC_BAD_SET_ARGUMENT message meaning that the set_field for the id is not correct	None yet

9.2.9 PushVlanVidPcp – FAILED

This test checks that a flow is correctly able to push a vlan in the header of a packet, to set the priority of the vlan (pcp), and to set the id of the vlan.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the push vlan, set pcp, and set vid instructions)
Pkt2	Dataplane ⇒ Switch	tcp packet (without vlan)
Pkt3	Switch ⇒ Dataplane	tcp packet (with vlan)
Tests		
	Check if a packet came out	

Error	Correction
The flow mod fails and the switch returns an OFPBAC_BAD_SET_ARGUMENT message meaning that the set_field for the id is not correct	None yet

9.2.10 SetEthDst – OK

This test checks that a flow is correctly able to change the Ethernet destination address in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the eth_dst instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.11 SetEthSrc – OK

This test checks that a flow is correctly able to change the Ethernet source address in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the eth_src instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.12 SetIPv4Dst – OK

This test checks that a flow is correctly able to change the IPV4 destination address in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ipv4_dst instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.13 SetIPv4Src – OK

This test checks that a flow is correctly able to change the IPV4 source address in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ipv4_src instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.14 SetIPv6Dst – OK

This test checks that a flow is correctly able to change the IPV6 destination address in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ipv6_dst instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.15 SetIPv6Flabel – OK

This test checks that a flow is correctly able to change the IPV6 flow label in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ipv6_flabel instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.16 SetIPv6Flabel_NonZeroDSCPandECN – OK

This test checks that a flow is correctly able to change the IPV6 flow label in the header of a tcp packet and that DSCP and ECN are not modified

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ipv6_flabel instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out Check the traffic class for errors on values	

9.2.17 SetIPv6Src – OK

This test checks that a flow is correctly able to change the IPV6 source address in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ipv6_src instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.18 SetIpv4DSCP_NonZeroECN – OK

This test checks that a flow is correctly able to change the IPV4 DSCP in the header of a tcp packet and that ECN is not modified

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_dscp instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out Check the type of service for errors on ECN	

9.2.19 SetIpv4Dscp – OK

This test checks that a flow is correctly able to change the IPV4 DSCP in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_dscp instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.20 SetIpv4ECN – OK

This test checks that a flow is correctly able to change the IPV4 ECN in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_ecn instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.21 SetIpv4ECN_NonZeroDSCP – OK

This test checks that a flow is correctly able to change the IPV4 ECN in the header of a tcp packet and that DSCP is not modified

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_ecn instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out Check the type of service for errors on DSCP	

9.2.22 SetIpv4TTL – OK

This test checks that a flow is correctly able to change the IPV4 TTL in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_ttl instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.23 SetIpv6DSCP_NonZeroECNandFlabel – OK

This test checks that a flow is correctly able to change the IPV6 DSCP in the header of a tcp packet and that ECN and flow labels are not modified.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_dscp instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out Check the type class for errors on values	

9.2.24 SetIpv6Dscp – OK

This test checks that a flow is correctly able to change the IPV6 DSCP in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_dscp instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.25 SetIpv6ECN – OK

This test checks that a flow is correctly able to change the IPV6 ECN in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_ecn instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.26 SetIpv6ECN_NonZeroDSCPandFlabel – OK

This test checks that a flow is correctly able to change the IPV6 ECN in the header of a tcp packet and that DSCP and flow labels are not modified.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_ecn instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out Check the type class for errors on values	

9.2.27 SetIpv6HopLimit – OK

This test checks that a flow is correctly able to change the IPV6 hop limit in the header of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the ip_hlim instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.28 SetTCPDst – OK

This test checks that a flow is correctly able to change the tcp destination port of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the tcp_dst instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.29 SetTCPDsrc – OK

This test checks that a flow is correctly able to change the tcp source port of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the tcp_src instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.30 SetUDPDst – OK

This test checks that a flow is correctly able to change the UDP destination port of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the udp_dst instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.31 SetUDPDsrc – OK

This test checks that a flow is correctly able to change the UDP source port of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the udp_src instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

9.2.32 SetVlanPcp – FAILED

This test checks that a flow is correctly able to change the vlan priority of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the vlan_pcp instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.2.33 SetVlanVid – FAILED

This test checks that a flow is correctly able to change the vlan id of a tcp packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPC_ADD (with the vlan_vid instruction)
Pkt2	Dataplane ⇒ Switch	tcp packet
Pkt3	Switch ⇒ Dataplane	tcp packet
Tests		
	Check if a packet came out	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.3 flow_mod

9.3.1 Overwrite – OK

This test checks that the controller can add a flow and then overwrite it without loose its stats

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPFC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPFC_ADD
Pkt3	Dataplane ⇒ Switch	tcp packet
Pkt4	Switch ⇒ Dataplane	tcp packet (checking stats)
Pkt5	Controller ⇒ Switch	Flow_Mod : OFPFC_ADD (with the same table_id, match, and priority, causing an overwrite)
Pkt6	Dataplane ⇒ Switch	tcp packet
Pkt7	Switch ⇒ Dataplane	tcp packet (checking stats)
Tests		
	Check stats were preserved after first flow add Check controller do not received any flow removed message from the switch Check stats were preserved after second flow add	

9.4 flow_stats

This test suite checks that the controller is capable of collecting the statistics from flows.

9.4.1 AllFlowStats – FAILED

This test checks that the controller can add three flows and get correct statistics from all of them.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPFC_ADD (with flag = OFPFF_NO_PKT_COUNTS)
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPFC_ADD (with flag = OFPFF_NO_BYT_COUNTS)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPFC_ADD (with flag = OFPFF_CHECK_OVERLAP)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Controller ⇒ Switch	Flow Stats request
Tests		
	Check the information received by controller are correct (table_id, priority, idle_timeout, hard_timeout, flags, cookie, actions) Check controller received information for all flows	

Error	Correction
OFPFF_CHECK_OVERLAP flag is not set properly (value is 2, switch set 0 for the flow). All other flags are correct	None yet

9.4.2 CookieFlowStats – OK

This test checks that the controller can add multiple flows with different cookies and mask.

Loop on cookies list		
Pkt1	Controller ⇒ Switch	Flow_Mod : OFPFC_ADD (with specific cookie)
Pkt2	Controller ⇒ Switch	barrier
Loop on cookies list		
Pkt3	Controller ⇒ Switch	Flow Stats request
Tests		
	Check for each combination of cookie and match, verify the correct flows are retrieved	

9.5 groups

This test suite checks all the functionalities related to the groups.

9.5.1 GroupAdd – OK (corrected)

This test checks that the controller can add a new group.

Pkt1 Pkt2	Controller ⇒ Switch Controller ⇒ Switch	Group_Mod : OFPGC_ADD group_desc_stats_request
Tests		
	Check the information received by controller are correct (group_type, group_id, actions)	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.2 GroupAddAllWeight – FAILED

This test checks that the controller can not add a new group with type = ALL and weights on the actions.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with weights)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_INVALID_GROUP	

Error	Correction
The switch accept the group and does not send an error	None

9.5.3 GroupAddExisting – OK (corrected)

This test checks that the controller can not add two groups with the same id.

Pkt1 Pkt2	Controller ⇒ Switch Controller ⇒ Switch	Group_Mod : OFPGC_ADD Group_Mod : OFPGC_ADD with same id
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_GROUP_EXISTS	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.4 GroupAddIndirectBuckets – OK

This test checks that the controller can not add an indirect group with more or less than one bucket.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with two buckets)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_INVALID_GROUP	

9.5.5 GroupAddIndirectWeight – FAILED

This test checks that the controller can not add an indirect group with a weight for the action.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (indirect group with weight)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_INVALID_GROUP	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.
The switch accept the group and does not send an error	None

9.5.6 GroupAddInvalidAction – FAILED

This test checks that the controller can not add a group with an invalid bucket.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with bad action)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_BAD_ACTION	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.
The switch accept the group and does not send an error	None

9.5.7 GroupAddInvalidID – OK (corrected)

This test checks that the controller can not add a group with an invalid ID

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with invalid ID)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_INVALID_GROUP	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.8 GroupAddMaxID – OK (corrected)

This test checks that the controller can add a group with ID = OFPG_MAX.

Pkt1 Pkt2 Pkt3	Controller ⇒ Switch Controller ⇒ Switch Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with id = OFPG_MAX) Barrier Group_Desc_Stats
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_INVALID_GROUP	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.9 GroupAddMinimumInvalidID – OK (corrected)

This test checks that the controller can add a group with ID = OFPG_MAX+1.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with id = OFPG_MAX+1)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_INVALID_GROUP	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.10 GroupAddSelectNoWeight – FAILED

This test checks that the controller can not add a select group with weights == 0.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with id = OFPG_MAX+1)
Tests		
	Check that the description received by the controller is correct	

Error	Correction
The switch accept the group and does not send an error	None

9.5.11 GroupDeleteAll – OK (corrected)

This test checks that the controller can add multiple groups and then delete them all.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with id = 0)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with id = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Controller ⇒ Switch	Group_Mod : OFPGC_DELETE
Pkt6	Controller ⇒ Switch	Group_Desc_Stats
Tests		
	Check that all the groups are deleted	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.12 GroupDeleteExisting – OK (corrected)

This test checks that the controller can add a group and then delete it.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with id = 0)
Pkt2	Controller ⇒ Switch	Barrier
Pkt5	Controller ⇒ Switch	Group_Mod : OFPGC_DELETE (with target group_id = 0)
Pkt6	Controller ⇒ Switch	Group_Desc_Stats
Tests		
	Check that the group is deleted	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.13 GroupDeleteNonexisting – OK

This test checks that the controller can delete a non existing group without triggering an error.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_DELETE (with target group_id = 0)
Tests		
	Check that there is no error	

9.5.14 GroupDescStats – OK (corrected)

This test checks that the controller can request type, id, and buckets for each groups with a group desc stats request.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (all group with multiple actions)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (select group with multiple actions)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (fast failover group with multiple actions)
Pkt6	Controller ⇒ Switch	Barrier
Pkt7	Controller ⇒ Switch	Group_Desc_Stats
Tests		
	Check that the information retrieved by the request are correct	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.15 GroupFlowSelect – OK

This test checks that the controller can request type, id, and buckets for each groups with a group desc stats request.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (group_id = 1, no action)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (group_id = 2, no action)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Controller ⇒ Switch	Flow_Mod : OFPFP_ADD (group 1)
Pkt6	Controller ⇒ Switch	Barrier
Pkt7	Controller ⇒ Switch	Flow_Mod : OFPFP_ADD (group 2)
Pkt8	Controller ⇒ Switch	Barrier
Pkt9	Controller ⇒ Switch	Flow_Mod : OFPFP_ADD (group 2))
Pkt10	Controller ⇒ Switch	Barrier
Pkt11	Controller ⇒ Switch	Flow_Mod : OFPFP_ADD (no group))
Pkt12	Controller ⇒ Switch	Barrier
Pkt13	Controller ⇒ Switch	aggregate_stats_request (for group 2)
Tests		
	Check that the numbler of flows is correct	

9.5.16 GroupModify – OK (corrected)

This test checks that the controller can add a group and then modify it.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (action = out-port1)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Group_Mod : OFPGC_MOD (action = out-port1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Controller ⇒ Switch	group_desc_stats_request
Tests		
	Check that the group has been modified	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.17 GroupModifyEmpty – OK (corrected)

This test checks that the controller can add a group and then modify it with an empty bucket.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (action = out-port1)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Group_Mod : OFPGC_MOD (no action)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Controller ⇒ Switch	group_desc_stats_request
Tests		
	Check that the group has been modified	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.18 GroupModifyInvalidID – OK (corrected)

This test checks that the controller can not modify a reserved group.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (with id = OFPG_ALL)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_INVALID_GROUP	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.19 GroupModifyLoop – OK (corrected)

This test checks that a modification causing a loop results in an error message.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (group0, with outport = port1)
Pkt2	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (group1, with outport = group0)
Pkt3	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (group2, with outport = group1)
Pkt4	Controller ⇒ Switch	Group_Mod : OFPGC_MOD (group0, with outport = group2)
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_LOOP	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.
OVS does not support group chaining	Nothing can be done

9.5.20 GroupModifyNonexisting – FAILED

This test checks that a modification for a non-existing group results in an error.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_MOD
Tests		
	Check that the controller receive an error message : OFPET_GROUP_MOD_FAILED, OFPGMFC_UNKNOWN_GROUP	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.21 GroupStats – OK (corrected)

This test checks that the controller can request the stats for the specified group.

Pkt1 Pkt2 Pkt3	Controller ⇒ Switch Controller ⇒ Switch Switch ⇒ Controller	Group_Mod : OFPGC_ADD group_stats_request OFPT_MULTIPART_REPLY : OFPMP_GROUP
Tests		
	Check that the controller receive the correct statistics (group_id, ref_count, packet_count, byte_count, bucket_stats)	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.22 GroupStatsAll – OK (corrected)

This test checks that the controller can request an entry for each group.

Pkt1 Pkt2 Pkt3 Pkt4 Pkt5 Pkt6	Controller ⇒ Switch Controller ⇒ Switch Controller ⇒ Switch Controller ⇒ Switch Controller ⇒ Switch Switch ⇒ Controller	Group_Mod : OFPGC_ADD Barrier Group_Mod : OFPGC_ADD Barrier group_stats_request OFPT_MULTIPART_REPLY : OFPMP_GROUP
Tests		
	Check that the controller receive the correct statistics (group_id, ref_count, packet_count, byte_count, bucket_stats)	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.23 GroupStatsNonExistent – OK

This test checks that the controller can request an entry for a non-existing group.

Pkt1	Controller ⇒ Switch	group_stats_request
Pkt2	Switch ⇒ Controller	OFPT_MULTIPART_REPLY : OFPMP_GROUP
Tests		
	Check that the controller receive an empty list	

9.5.24 SelectFwdEmpty – OK (corrected)

This test checks that a select group without any bucket does not alter the action set on the packet.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (select group, no bucket)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	tcp packet
Pkt6	Switch ⇒ Dataplanes	tcp packet
Tests		
	Check that the dataplanes correctly received the tcp packet fowarded by the switch	

Error	Correction
The output port was not set to a correct value in the flow, therefore the packet was dropped.	The port was set to a correct value

9.5.25 SelectFwdSingle – OK (corrected)

This test checks that a packet forwarded by a flow to a select group with one action, should use that action.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (select group, one bucket)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (forward towards the group)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	tcp packet
Pkt6	Switch ⇒ Dataplanes	tcp packet
Tests		
	Check that the dataplanes correctly received the tcp packet fowarded by the switch	

Error	Correction
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.5.26 SelectFwdSpread – OK (corrected)

This test checks that a select group with several buckets correctly spreads different flows between them.

Pkt1	Controller ⇒ Switch	Group_Mod : OFPGC_ADD (select group, several buckets)
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (forward towards the group)
Pkt4	Controller ⇒ Switch	Barrier
Loop on 1000 packets		
Pkt5	Dataplanes ⇒ Switch	tcp packet
Pkt6	Switch ⇒ Dataplanes	tcp packet
Tests		
	Check that the number of packets which go through each port is within 20% of expected.	

Error	Correction
Test was using 4 ports, but only 3 are available on the switch.	Test was modified to use only 3 ports.
The watch_port should be set to OFPP_ANY, and watch_group should be set to OFPG_ANY when the fields are to be ignored. However the current ofp_bucket record definition default these fields to zero.	when they are to be ignored, those field are now set to the correct value.

9.6 match

This test suite checks that the flows are capable of matching all the different types of field.

9.6.1 ArpOp – OK

This test checks that a flow is capable of matching an ARP operation.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	arp packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	arp packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	arp packet (with wrong matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching arp packet is correctly fowarded and that a non-matching arp packet is correctly sent to the controller via packetIn	

9.6.2 ArpSPA – OK

This test checks that a flow is capable of matching the sender IP of an ARP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	arp packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	arp packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	arp packet (with wrong matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching arp packet is correctly fowarded and that a non-matching arp packet is correctly sent to the controller via packetIn	

9.6.3 ArpSPAMasked – OK

This test checks that a flow is capable of matching the sender IP arbitrarily masked of an ARP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	arp packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	arp packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	arp packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching arp packet is correctly fowarded and that a non-matching arp packet is correctly sent to the controller via packetIn	

9.6.4 ArpSPASubnetMasked – OK

This test checks that a flow is capable of matching the sender IP with a subnet mask of an ARP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	arp packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	arp packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	arp packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching arp packet is correctly fowarded and that a non-matching arp packet is correctly sent to the controller via packetIn	

9.6.5 ArpTPA – OK

This test checks that a flow is capable of matching the target IP of an ARP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	arp packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	arp packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	arp packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching arp packet is correctly fowarded and that a non-matching arp packet is correctly sent to the controller via packetIn	

9.6.6 ArpTPAMasked – OK

This test checks that a flow is capable of matching the target IP arbitrarily masked of an ARP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	arp packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	arp packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	arp packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching arp packet is correctly fowarded and that a non-matching arp packet is correctly sent to the controller via packetIn	

9.6.7 ArpTPASubnetMasked – OK

This test checks that a flow is capable of matching the target IP with a subnet mask of an ARP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	arp packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	arp packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	arp packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching arp packet is correctly fowarded and that a non-matching arp packet is correctly sent to the controller via packetIn	

9.6.8 EthDst – OK

This test checks that a flow is capable of matching the ethernet destination of a TCP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt6	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

9.6.9 EthDstBroadcast – OK

This test checks that a flow is capable of matching the ethernet destination (broadcast) of a TCP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt6	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

9.6.10 EthDstMasked – OK

This test checks that a flow is capable of matching the masked ethernet destination of a TCP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt6	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

9.6.11 EthDstMulticast – OK

This test checks that a flow is capable of matching the masked ethernet destination (IPV4 multicast) of a TCP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt6	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

9.6.12 EthSrc – FAILED

This test checks that a flow is capable of matching the ethernet source of a TCP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt6	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

Error	Correction
Packets are not matched properly	None yet
Test is making matching on TCP headers impossible for 5min	Test deactivated

9.6.13 EthSrcBroadcast – FAILED

This test checks that a flow is capable of matching the ethernet source (broadcast) of a TCP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt10	Switch ⇒ Controller	packetIn
Pkt11	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt12	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

Error	Correction
Packets are not matched properly	None yet
Test is making matching on TCP headers impossible for 5min	Test deactivated

9.6.14 EthSrcMasked – FAILED

This test checks that a flow is capable of matching on the ethernet source (masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt8	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt9	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt10	Switch ⇒ Controller	packetIn
Pkt11	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt12	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

Error	Correction
Packets are not matched properly	None yet
Test is making matching on TCP headers impossible for 5min	Test deactivated

9.6.15 EthSrcMulticast – FAILED

This test checks that a flow is capable of matching the ethernet source (broadcast) of a TCP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with wrong matching matching information)
Pkt8	Dataplanes ⇒ Switch	TCP packet (with wrong matching matching information)
Pkt9	Dataplanes ⇒ Switch	TCP packet (with wrong matching matching information)
Pkt10	Dataplanes ⇒ Switch	TCP packet (with wrong matching matching information)
Pkt11	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching TCP packet is correctly fowarded and that a non-matching TCP packet is correctly sent to the controller via packetIn	

Error	Correction
Packets are not matched properly	None yet
Test is making matching on TCP headers impossible for 5min	Test deactivated

9.6.16 EthTypeARP – OK

This test checks that a flow is capable of matching the ethertype (0x0806) of an ARP packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	ARP packet (with matching information)
Pkt6	Switch ⇒ Dataplanes	ARP packet (with matching information)
Pkt5	Dataplanes ⇒ Switch	ARP packet (without matching information)
Pkt6	Switch ⇒ Controller	packetIn
Tests		
	Check that a matching ARP packet is correctly fowarded and that a non-matching ARP packet is correctly sent to the controller via packetIn	

9.6.17 EthTypeIPv4 – OK (corrected)

This test checks that a flow is capable of matching the ethertype (IPV4) of a tcp packet, or an udp packet or an icmp packet or a snap packet, but rejects an arp packet or an ipv6 packet or a llc packet..

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet
Pkt6	Dataplanes ⇒ Switch	UDP packet
Pkt7	Dataplanes ⇒ Switch	ICMP packet
Pkt8	Dataplanes ⇒ Switch	snap packet
Pkt9	Dataplanes ⇒ Switch	ARP packet
Pkt10	Switch ⇒ Controller	packetIn
Pkt11	Dataplanes ⇒ Switch	LLC packet
Pkt12	Switch ⇒ Controller	packetIn
Pkt13	Dataplanes ⇒ Switch	ipv6 packet
Pkt14	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
Test was also testing the matching of a tcp packet tagged with a Vlan, but as our switch is using his own vlan, the tag was scrapped.	the tagged packet was removed

9.6.18 EthTypeIPv6 – OK (corrected)

This test checks that a flow is capable of matching the ethertype (IPV6) of a tcp ipv6 packet, or an udp ipv6 packet or an icmp ipv6 packet, but rejects an arp packet or an ipv4 packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet
Pkt6	Dataplanes ⇒ Switch	UDP ipv6 packet
Pkt7	Dataplanes ⇒ Switch	ICMP ipv6 packet
Pkt8	Dataplanes ⇒ Switch	tcp ipv4 packet
Pkt9	Switch ⇒ Controller	packetIn
Pkt10	Dataplanes ⇒ Switch	ARP packet
Pkt11	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
Test was also testing the matching of a tcp packet tagged with a Vlan, but as our switch is using his own vlan, the tag was scrapped.	the tagged packet was removed

9.6.19 EthTypeNone – OK

This test checks that a flow is capable of matching on no ethertype (IEEE 802.3 without SNAP header) of an llc packet, but rejects a tcp packet or an ipv6 packet or a snap packet.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet
Pkt6	Dataplanes ⇒ Switch	UDP ipv6 packet
Pkt7	Dataplanes ⇒ Switch	ICMP ipv6 packet
Pkt8	Dataplanes ⇒ Switch	tcp ipv4 packet
Pkt9	Switch ⇒ Controller	packetIn
Pkt10	Dataplanes ⇒ Switch	tcp ipv6 packet
Pkt11	Switch ⇒ Controller	packetIn
Pkt12	Dataplanes ⇒ Switch	tcp snap packet
Pkt13	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.20 IPv4Dscp – OK

This test checks that a flow is capable of matching on ipv4 dscp.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information (dscp = 4), output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (dscp=4 ecn=0)
Pkt6	Dataplanes ⇒ Switch	TCP packet (dscp=4 ecn=3)
Pkt7	Dataplanes ⇒ Switch	TCP packet (dscp=5 ecn=0)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.21 IPv4Dst – OK

This test checks that a flow is capable of matching on ipv4 destination address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.22 IPv4DstMasked – OK

This test checks that a flow is capable of matching on masked ipv4 destination address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.23 IPv4DstSubnetMasked – OK

This test checks that a flow is capable of matching on ipv4 destination address (subnet masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt8	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt9	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt10	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt11	Switch ⇒ Controller	packetIn
Pkt12	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt13	Switch ⇒ Controller	packetIn
Pkt14	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt15	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.24 IPv4Ecn – OK

This test checks that a flow is capable of matching on ipv4 ecn.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.25 IPv4ICMPCode – OK

This test checks that a flow is capable of matching on ipv4 icmp code.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.26 IPv4ICMPType – OK

This test checks that a flow is capable of matching on ipv4 icmp type.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.27 IPv4ProtoICMP – OK

This test checks that a flow is capable of matching on protocol field (ICMP).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.28 IPv4ProtoTCP – OK

This test checks that a flow is capable of matching on ipv4 protocol field (TCP).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.29 IPv4ProtoUDP – OK

This test checks that a flow is capable of matching on ipv4 protocol field (UDP).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.30 IPv4Src – OK

This test checks that a flow is capable of matching on ipv4 source address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.31 IPv4SrcMasked – OK

This test checks that a flow is capable of matching on a masked ipv4 source address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.32 IPv4SrcSubnetMasked – OK

This test checks that a flow is capable of matching on a ipv4 source address (subnet masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt8	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt9	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt10	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt11	Switch ⇒ Controller	packetIn
Pkt12	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt13	Switch ⇒ Controller	packetIn
Pkt14	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt15	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.33 IPv4TCPDst – OK

This test checks that a flow is capable of matching on a ipv4 tcp destination port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.34 IPv4TCPDstMasked – OK

This test checks that a flow is capable of matching on a ipv4 tcp destination port (masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.35 IPv4TCPSrc – OK

This test checks that a flow is capable of matching on a ipv4 tcp source port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.36 IPv4TCPSrcMasked – OK

This test checks that a flow is capable of matching on a ipv4 tcp source port (masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.37 IPv4UDPDst – OK

This test checks that a flow is capable of matching on a ipv4 UDP destination port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	UDP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.38 IPv4UDPDstMasked – OK

This test checks that a flow is capable of matching on a ipv4 UDP destination port (masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	UDP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	UDP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.39 IPv4UDPSrc – OK

This test checks that a flow is capable of matching on a ipv4 UDP source port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	UDP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.40 IPv4UDPSrcMasked – OK

This test checks that a flow is capable of matching on a ipv4 UDP source port (masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	UDP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	UDP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	UDP packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.41 IPv6Dscp – OK

This test checks that a flow is capable of matching on a ipv6 dscp.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.42 IPv6Dst – OK

This test checks that a flow is capable of matching on ipv6 destination address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.43 IPv6DstMasked – OK

This test checks that a flow is capable of matching on a masked ipv6 destination address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Pkt10	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt11	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.44 IPv6DstSubnetMasked – OK

This test checks that a flow is capable of matching on a ipv6 destination address (subnet masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.45 IPv6Ecn – OK

This test checks that a flow is capable of matching on a ipv6 ecn.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.46 IPv6ICMPCode – OK

This test checks that a flow is capable of matching on a ipv6 icmp code.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.47 IPv6ICMPType – OK

This test checks that a flow is capable of matching on a ipv6 icmp type.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.48 IPv6ProtoICMP – OK

This test checks that a flow is capable of matching on a ipv6 protocol field (ICMP).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	icmp ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	udp ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.49 IPv6ProtoTCP – OK

This test checks that a flow is capable of matching on a ipv6 protocol field (TCP).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	udp ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	icmp ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.50 IPv6ProtoUDP – OK

This test checks that a flow is capable of matching on a ipv6 protocol field (UDP).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	udp ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	icmp ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.51 IPv6Src – OK

This test checks that a flow is capable of matching on ipv6 source address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.52 IPv6SrcMasked – OK

This test checks that a flow is capable of matching on a masked ipv4 source address.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Pkt10	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt11	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.53 IPv6SrcSubnetMasked – OK

This test checks that a flow is capable of matching on a ipv4 source address (subnet masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.54 IPv6TCPDst – OK

This test checks that a flow is capable of matching on a ipv6 tcp destination port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.55 IPv6TCPSrc – OK

This test checks that a flow is capable of matching on a ipv6 tcp source port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.56 IPv6UDPDst – OK

This test checks that a flow is capable of matching on a ipv6 UDP destination port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	UDP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	UDP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	UDP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.57 IPv6UDPSrc – OK

This test checks that a flow is capable of matching on a ipv6 UDP source port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	UDP ipv6 packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	UDP ipv6 packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	UDP ipv6 packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.58 InPort – OK

This test checks that a flow is capable of matching on ingress port.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without matching information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.6.59 VlanAbsent – FAILED

This test checks that a flow is capable of matching on absent Vlan tag.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (without Vlan)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with Vlan information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (with Vlan information)
Pkt9	Switch ⇒ Controller	packetIn
Pkt10	Dataplanes ⇒ Switch	TCP packet (with Vlan information)
Pkt11	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.6.60 VlanExact – FAILED

This test checks that a flow is capable of matching on absent Vlan ID and PCP.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt9	Switch ⇒ Controller	packetIn
Pkt10	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt11	Switch ⇒ Controller	packetIn
Pkt12	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt13	Switch ⇒ Controller	packetIn
Pkt14	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt15	Switch ⇒ Controller	packetIn
Pkt16	Dataplanes ⇒ Switch	TCP packet (without Vlan information)
Pkt17	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.6.61 VlanPCP – FAILED

This test checks that a flow is capable of matching on absent Vlan PCP (VID matched).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt7	Switch ⇒ Controller	packetIn
Pkt8	Dataplanes ⇒ Switch	TCP packet (without Vlan information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.6.62 VlanPCPAnyVID – FAILED

This test checks that a flow is capable of matching on absent Vlan PCP (VID absent).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (without Vlan information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.6.63 VlanPCPMasked – FAILED

This test checks that a flow is capable of matching on VLAN PCP (masked, VID matched).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt10	Switch ⇒ Controller	packetIn
Pkt11	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt12	Switch ⇒ Controller	packetIn
Pkt13	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt14	Switch ⇒ Controller	packetIn
Pkt15	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt16	Switch ⇒ Controller	packetIn
Pkt17	Dataplanes ⇒ Switch	TCP packet (without Vlan information)
Pkt18	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.6.64 VlanPresent – FAILED

This test checks that a flow is capable of matching on any VLAN tag (but must be present).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt8	Dataplanes ⇒ Switch	TCP packet (without Vlan information)
Pkt9	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.6.65 VlanVID – FAILED

This test checks that a flow is capable of matching on VLAN VID.

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (without Vlan information)
Pkt10	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.6.66 VlanVIDMasked – FAILED

This test checks that a flow is capable of matching on VLAN VID (masked).

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (with matching information, output = port, priority = 1000)
Pkt3	Controller ⇒ Switch	Flow_Mod : OFPfC_ADD (wildcard, output = controller, priority = 1)
Pkt4	Controller ⇒ Switch	Barrier
Pkt5	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt6	Dataplanes ⇒ Switch	TCP packet (with matching information)
Pkt7	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt8	Switch ⇒ Controller	packetIn
Pkt9	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt10	Switch ⇒ Controller	packetIn
Pkt11	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt12	Switch ⇒ Controller	packetIn
Pkt13	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt14	Switch ⇒ Controller	packetIn
Pkt15	Dataplanes ⇒ Switch	TCP packet (with wrong matching information)
Pkt16	Switch ⇒ Controller	packetIn
Pkt17	Dataplanes ⇒ Switch	TCP packet (without Vlan information)
Pkt18	Switch ⇒ Controller	packetIn
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

9.7 role_request

A controller can have different roles (master, equal, slave), this test suite checks that it can navigate between the different roles with the correct rights.

9.7.1 RolePermissions – OK

This test checks that a flow is capable of matching on VLAN VID (masked).

verify_permission(perm)		
Pkt1	Controller ⇒ Switch	Packet_out
Pkt2	Controller ⇒ Switch	Flow_Mod : OFPFC_DELETE
Pkt3	Controller ⇒ Switch	Group_Mod : OFPGC_DELETE
Pkt4	Controller ⇒ Switch	Barrier
Loop on three packets		
Pkt5	Controller ⇒ Switch	Packet_out
Tests		
	if perm == true, check that there is no error (controller have permission), if perm == false, check there are three errors (controller does not have permission)	

Pkt1	Controller ⇒ Switch	Flow_Mod : OFPfC_DELETE
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Function	verify_permission(True)
Pkt4	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt5	Switch ⇒ Controller	OFPT_ROLE_Reply
Pkt6	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER
Pkt7	Switch ⇒ Controller	OFPT_ROLE_Reply
Pkt8	Function	verify_permission(True)
Pkt9	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt10	Switch ⇒ Controller	OFPT_ROLE_Reply
Pkt11	Function	verify_permission(FALSE)
Pkt12	Controller ⇒ Switch	OFPT_ROLE_Request (OFPCR_ROLE_EQUAL)
Pkt13	Switch ⇒ Controller	OFPT_ROLE_Reply
Pkt14	Function	verify_permission(TRUE)
Tests		
	Check that all packets which should be forwarded are indeed forwarded and that those which should not are correctly dropped	

9.7.2 RoleRequestEqualToMaster – OK (corrected)

This test checks that a controller is capable of switching between equal and master roles and back.

Pkt1	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt2	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt3	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER
Pkt4	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt5	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_EQUAL
Pkt6	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt7	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER with smallest greater generation ID
Pkt8	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt9	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_EQUAL
Pkt10	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt11	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER with largest greater generation ID
Pkt12	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt13	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_EQUAL
Pkt14	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt13	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER
Pkt14	Switch ⇒ Controller	OFPT_ERROR
Pkt15	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt16	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt17	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER
Pkt18	Switch ⇒ Controller	OFPT_ERROR
Pkt19	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt20	Switch ⇒ Controller	OFPT_ROLE_REPLY
Tests		
	Check that the role of the controller remains unchanged when it sends a request for master role with least stale generation	
	Check that the role of the controller remains unchanged when it sends a request for master role with most stale generation	

Error	Correction
The specification are wrongly implemented on OVS, forcing the genID to be specified when setting the role equal (this should only be forced for master and slave)	we are allowing the test to be passed by adding the genID in the message (with –correction) for test purposes, but the intended goal of the library is to spot those errors, and therefore this test should not be passed

9.7.3 RoleRequestEqualToSlave – OK (corrected)

This test checks that a controller is capable of switching between equal and slave roles and back.

Pkt1	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt2	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt3	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt4	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt5	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_EQUAL
Pkt6	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt7	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE with smallest greater generation ID
Pkt8	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt9	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_EQUAL
Pkt10	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt11	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE with largest greater generation ID
Pkt12	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt13	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_EQUAL
Pkt14	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt13	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt14	Switch ⇒ Controller	OFPT_ERROR
Pkt15	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt16	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt17	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt18	Switch ⇒ Controller	OFPT_ERROR
Pkt19	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt20	Switch ⇒ Controller	OFPT_ROLE_REPLY
Tests		
	Check that the role of the controller remains unchanged when it sends a request for slave role with least stale generation	
	Check that the role of the controller remains unchanged when it sends a request for slave role with most stale generation	

Error	Correction
The specification are wrongly implemented on OVS, forcing the genID to be specified when setting the role equal (this should only be forced for master and slave)	we are allowing the test to be passed by adding the genID in the message (with –correction) for test purposes, but the intended goal of the library is to spot those errors, and therefore this test should not be passed

9.7.4 RoleRequestNoChange – OK

This test checks that a controller is capable of querying for its current role and generation id.

Pkt1	Controller ⇒ Switch	OFPT_ROLE_Request no change (save generation id in gen)
Pkt2	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt3	Controller ⇒ Switch	OFPT_ROLE_Request no change (save generation id in new_gen)
Pkt4	Switch ⇒ Controller	OFPT_ROLE_REPLY
Tests		
	Check that the role of the controller is "Equal"	
	Check that generation id does not change between two requests	

9.7.5 RoleRequestSlaveToMaster – OK

This test checks that a controller is capable of switching between equal and slave roles and back.

Pkt1	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt2	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt3	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt4	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt5	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER
Pkt6	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt7	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt8	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt9	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER with smallest greater generation ID
Pkt10	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt11	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt12	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt13	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER with largest greater generation ID
Pkt14	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt15	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt16	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt17	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER (with least stale generation ID)
Pkt18	Switch ⇒ Controller	OFPT_ERROR
Pkt19	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt20	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt22	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER (with most stale generation ID)
Pkt23	Switch ⇒ Controller	OFPT_ERROR
Pkt24	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt25	Switch ⇒ Controller	OFPT_ROLE_REPLY
Tests		
	Check that the role of the controller remains unchanged when it sends a request for master role with least stale generation	
	Check that the role of the controller remains unchanged when it sends a request for master role with most stale generation	

9.7.6 SlaveNoPacketIn – OK

This test checks that slave connections do not receive OFPT_PACKET_IN messages but other roles do.

Pkt1	Controller ⇒ Switch	OFPT_FLOW_MOD OFPFC_DELETE
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Switch ⇒ Controller	Barrier reply
Pkt4	Controller ⇒ Switch	OFPT_FLOW_MOD OFPFC_ADD
Pkt5	Switch ⇒ Controller	Barrier
Pkt6	Switch ⇒ Controller	Barrier reply
Pkt7	Controller ⇒ Switch	OFPT_ROLE_Request no change
Pkt8	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt9	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_MASTER
Pkt10	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt11	Controller ⇒ Switch	tcp packet
Pkt12	Switch ⇒ Controller	OFPT_PACKET_IN
Pkt13	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_SLAVE
Pkt14	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt15	Controller ⇒ Switch	tcp packet
Pkt16	Controller ⇒ Switch	OFPT_ROLE_Request OFPCR_ROLE_EQUAL
Pkt17	Switch ⇒ Controller	OFPT_ROLE_REPLY
Pkt18	Controller ⇒ Switch	tcp packet
Pkt19	Switch ⇒ Controller	OFPT_PACKET_IN
Tests		
	Check that slave connections do not receive OFPT_PACKET_IN messages but other roles do	

9.8 pktin_match

This test suite checks that the controller correctly receives a pktin message when the switch receives a message and it has a flow redirecting the messages to the controller.

9.8.1 VlanAbsent – OK

This test checks that the controller correctly received a packetIn message triggered by a tcp packet without a vlan tag.

Pkt1	Controller ⇒ Switch	OFPT_FLOW_MOD OFPFC_DELETE
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Switch ⇒ Controller	Barrier reply
Pkt4	Controller ⇒ Switch	OFPT_FLOW_MOD OFPFC_ADD
Pkt5	Switch ⇒ Controller	Barrier
Pkt6	Switch ⇒ Controller	Barrier reply
Pkt7	Dataplanes ⇒ Switch	tcp packet
Pkt8	Switch ⇒ Controller	OFPT_PACKET_IN
Tests		
	Check that the controller received the packetIn message.	

9.8.2 VlanVid – FAILED

This test checks that the controller correctly received a packetIn message triggered by a tcp packet with a vlan tag.

Pkt1	Controller ⇒ Switch	OFPT_FLOW_MOD OFPFC_DELETE
Pkt2	Controller ⇒ Switch	Barrier
Pkt3	Switch ⇒ Controller	Barrier reply
Pkt4	Controller ⇒ Switch	OFPT_FLOW_MOD OFPFC_ADD
Pkt5	Switch ⇒ Controller	Barrier
Pkt6	Switch ⇒ Controller	Barrier reply
Pkt7	Dataplanes ⇒ Switch	tcp packet
Pkt8	Switch ⇒ Controller	OFPT_PACKET_IN
Tests		
	Check that the controller received the packetIn message.	

Error	Correction
The switch is using vlans to handle his physical ports and thus is scrapping the vlan parts from the packet when they arrived, this causes the flow to not be able to match the packet, therefore the packet is dropped	None yet

10 Bsn messages

Big Switch Network is a company that support the openflood project, the bsn tests are used to test openflow related messaged for the switchs of this company. As we are using a TP-Link, we can not test nor document those tests.